2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A274 942

DTIC
S ELECTE
JAN 2 6 1994
A

THESIS

NPSNET-IV :
A Real-Time, 3D Distributed
Interactive Virtual World

by

Roy David Young

September 1993

Thesis Advisor:	Michael J. Zyda
Co-Advisor:	David R. Pratt

Approved for public release; distribution is unlimited.

94-02156

94 1 25 054

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis, July 1991 - September 1993

4. TITLE AND SUBTITLE NPSNET-IV: A Real-Time, 3D Distributed Interactive Virtual World	5. FUNDING NUMBERS
6. AUTHOR(S) Young, Roy David	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-500	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Unclassified/Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The problems addressed by this research are to develop a new NPSNET simulator to allow simulations with any simulator that complies with the Distributed Interactive Simulation (DIS) protocol; to provide a more realistic simulation by maximizing the use of the Silicon Graphics Inc. (SGI) Reality Engine; and to provide for future extensions to the system.The approach taken for this research was to develop a DIS based simulator. To provide greater realism, Performer, a 3-D toolkit made by SGI was used to take advantage of its multiprocessing management capabilities, real-time scene management, and other rapid rending tools. To enhance the systems extensibility, C++ object classes were used to encapsulate entity behaviors and user inputs.The result of this work is the NPSNET-IV simulation system. This system uses the DIS protocol to interact with other heterogeneously developed simulators as was demonstrated in a week long simulation between NPSNET-IV and two different simulation systems written by the Air Force Institute of Technology. Performer's direct access to the Reality Engine hardware and real-time scene management allows integration of more realistic models and higher rates of movement in the virtual world. The extensibility of the system is enhanced through the use of C++ objects, which was proven by the addition of a submersible vehicle type to the simulation.

14. SUBJECT TERMS NPSNET, DIS, real-time, 3D, visual simulation, network, distributed, Performer, autonomous forces, interactive, virtual world, stereo graphics	15. NUMBER OF PAGES 77
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

NPSNET-IV : A Real-Time, 3D Distributed Interactive Virtual World

by
Roy David Young
Captain, United States Marine Corps
B.S., Southern Illinois University, 1987

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Author: Roy David Young
Roy David Young

Approved By: [illegible]
Dr. Michael J. Zyda, Thesis Advisor

[illegible]
Dr. David R. Pratt, Thesis Co-Advisor

Ted Lewis
Dr. Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The problems addressed by this research are to develop a new NPSNET simulator to allow simulations with any simulator that complies with the Distributed Interactive Simulation (DIS) protocol; to provide a more realistic simulation by maximizing the use of the Silicon Graphics Inc. (SGI) Reality Engine; and to provide for future extensions to the system. The approach taken for this research was to develop a DIS based simulator. To provide greater realism, Performer, a 3-D toolkit made by SGI was used to take advantage of its multiprocessing management capabilities, real-time scene management, and other rapid rending tools. To enhance the systems extensibility, C++ object classes were used to encapsulate entity behaviors and user inputs. The result of this work is the NPSNET-IV simulation system. This system uses the DIS protocol to interact with other heterogeneously developed simulators as was demonstrated in a week long simulation between NPSNET-IV and two different simulation systems written by the Air Force Institute of Technology. Performer's direct access to the Reality Engine hardware and real-time scene management allows integration of more realistic models and higher rates of movement in the virtual world. The extensibility of the system is enhanced through the use of C++ objects, which was proven by the addition of a submersible vehicle type to the simulation.

DTIC QUALITY INSPECTED 5

Accesion For		
NTIS CRA&I	☑	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

B. FOCUS 2

C. SUMMARY OF CHAPTERS 3

II. OVERVIEW OF NPSNET-IV 4

A. MODELING AND GEOMETRY 4

B. SIMULATION CODE 6

C. NETWORKING 7

D. AUTONOMOUS AGENTS 8

E. SPATIALIZED SOUND 8

III. VISUAL DATABASE DESIGN AND DEVELOPMENT 11

A. HIERARCHICAL DATABASE STRUCTURE 11

1. MultiGen Flight 11

a. Database (DB) Bead 12

b. Group Beads 12

c. Level-of-detail (LOD) Beads 12

d. External Reference Beads 14

e. Instance Beads 14

f. Object Beads 14

g. Polygon Beads 15

h. Generic Beads 15

2. Performer 15

a. pfScene 16

b. pfGroup 17

c. pfSCS 17

d. pfDCS 17

e. pfLayer 17
f. pfSwitch 18
g. pfLOD 18
h. pfSequence 18
i. pfGeode 18
j. pfBillboard 18
k. pfLightPoint 19
B. FLIGHT TO PERFORMER CONVERSION 19
C. BUILDING AN ANIMATED SEQUENCE 20
IV. PERFORMER OVERVIEW 22
A. MULTIPROCESSING AND SHARED DATA MANAGEMENT 22
1. Performer Processes 22
a. Application Process 25
b. Cull Process 26
c. Draw Process 27
2. Process Data Communication 28
a. Pass-through Data 28
b. Shared Data 29
c. pfDataPool 29
B. VISUAL EFFECTS SUPPORT 29
1. Earth and Sky Model 29
a. Fog 30
b. Clouds 30
2. Light Points 31
C. MATHEMATICS SUPPORT ROUTINES 31
1. Vectors 31
2. 4 x 4 Matrices 32
3. Volumes 32

D. INTERSECTION TESTING 32
V. IMPLEMENTATION 34
A. C++ OBJECT ORIENTED PARADIGM 34
1. Entities 34
2. Controls 36
B. CONFIGURATION FILES 36
C. INTERFACE OPTIONS 37
D. DATA STRUCTURES 38
1. Pass-through Data 38
2. Shared Data 39
3. Global Data 39
E. CALL BACKS 40
1. Drawing Routine 41
2. Intersection Testing 41
a. Collision Detection 41
b. Vehicle Control 42
VI. AUTONOMOUS AGENTS 43
A. INTRODUCTION TO CLIPS 43
B. NPSNET AUTONOMOUS VEHICLE CONTROL CLIPS SHELL 44
1. 'C' to 'C++' interface 45
2. System Control Rules 46
3. Entity Control Rules 46
VII. NETWORK IMPLEMENTATION 47
A. BACKGROUND 47
B. DIS PROTOCOL 47
C. PLAYERS AND GHOSTS 48
D. IMPLEMENTING DIS IN NPSNET-IV 49
1. Entity State PDUs 51

a. Sending 51
b. Receiving 52
2. Fire PDUs 52
a. Sending 53
b. Receiving 53
3. Detonation PDUs 53
a. Send 53
b. Receive 54
VIII. RESULTS, CONCLUSIONS, AND FUTURE WORK 55
A. RESULTS 55
1. Performance 55
2. Development and Extensibility 56
B. CONCLUSIONS 56
C. FUTURE WORK 57
APPENDIX A. NPSNET IV USER'S GUIDE 58
LIST OF REFERENCES 64
INITIAL DISTRIBUTION LIST 66

LIST OF FIGURES

Figure 1 A Simple World Model 5
Figure 2 Logical versus Spatial Organization 6
Figure 3 NPSSSS Speaker Placement 9
Figure 4 Additive LODs 13
Figure 5 Replacement LODs 13
Figure 6 Performer Node Hierarchy 16
Figure 7 Processes and Data Flow in the Performer Environment 23
Figure 8 Single and Triple Thread Latency 24
Figure 9 MUNITION Class Hierarchy 35
Figure 10 VEHICLE Class Hierarchy 35
Figure 11 NPSNET-IV Heads Up Display (HUD) 38
Figure 12 NPSNET-IV PASS_DATA 39
Figure 13 CLIPS Block Diagram 44
Figure 14 DIS and Performer Coordinate Systems 50
Figure 15 Heading Conversions 51

ACKNOWLEDGEMENTS

This project is not the work of one individual alone. There are a large number of people who contributed to this work both directly and indirectly. I would like to take this opportunity to thank a few who's help and guidance was invaluable in making NPSNET-IV a (virtual) reality.

Thanks, Dr. Michael Zyda for taking me on as a thesis student. I appreciate your approach to business, and thank you for your support throughout this work.

Thanks, Dr. David Pratt for your technical guidance, coding abilities, and moral support regardless of the hour. Most of all thanks for your patience. This task would have been impossible without you and I would probably still be looking for a thesis topic on graduation day. SIGGRAPH was fun, lets not do it again sometime.

Thanks, Mr. Paul Barham for the brain-storm sessions which lead to the resolution of several problems including getting the code to operate on multiple processes, the successful conversion to C++, and some great ways to improve the user interface. Your anal programming practices were inspiring, and helped make SIGGRAPH a great success.

Thanks, LT. John Goetz for all of your help with the autonomous vehicle control code. You really should be in the graphics track, you definitely missed your calling! Your enthusiasm and drive was a tremendous help in making the last two quarters bearable.

I would also like to thank all of my classmates in the Graphics Track for their constant willingness to share ideas and provide recommendations for improvement, especially Dennis Schmidt and Dan Corbin for their continual assistance and moral support in development of this work.

Finally, I would like to thank my wife, Sue, my daughter ShawnnaLea, and my son Ryan, for their loving support and understanding during the many nights and weekends I spent at the Graphics Lab.

Without the help and moral support of the above people and countless others, this work would not have been possible.

I. INTRODUCTION

A. BACKGROUND

NPSNET is a real-time, three-dimensional (3-D), distributed interactive virtual world being continually developed by researchers and students in the Graphics and Video Laboratory of the Department of Computer Science at the Naval Postgraduate School. It began as a low-cost, workstation based simulator which used a locally developed network protocol and ran on commercially available Silicon Graphic Inc. (SGI) workstations. Several follow-on versions with a wide range of differing capabilities have been developed. These include NPSStealth which displays ongoing or pre-recorded SIMNET simulations in a 3-D format, allowing the viewer free movement within the simulation without interaction. Today, NPSNET has evolved into a diverse family of simulators, expert systems, and networking related code all with the common goal of developing a fully-interactive and realistic simulation environment.

Over the past few years, the simulation system known as NPSNET has continued to evolve as numerous students and faculty members have developed and added functionality. As the functionality increased, the display rate of the system decreased. With textured terrain and 500 entities, the system now runs at 6 to 8 frames per second [Zyda92], which is too slow for an effective virtual world system [Brys93].

Numerous applications and technologies that have been developed as a result of NPSNET research are in widespread use today by many government and government sponsored organizations [Zyda92]. Despite its obvious success, NPSNET-I, like any large evolving software system is becoming a maintainability nightmare [Broo75]. Without a major re-engineering, the code can not effectively use many of the hardware improvements that are now available with the Reality Engine series workstations.

Most of the code developed by NPSNET research to date has been written in the C language. One of the research group's goals is to change to an object oriented paradigm using C++ in order to enhance the extensibility of the system for the future. NPSNET-III was the first attempt toward this goal, but never reached an operational state. NPSNET- IV is the second attempt, it is written in C++ and uses class inheritance concepts for its entity behaviors.

NPSNET-II was a version that added collision detection, munition trajectory, and terrain paging to the simulation system.The primary limiting factor with it, is the speed at which you can traverse the virtual world. Although the size of the terrain model is limited only by disk space, the system only keeps a 16 by 16 kilometer area in memory at a time. Due to the time required to page upcoming terrain into memory, the speed at which you could travel across the world without hesitations in the simulation is limited. These limitations prevent fast moving, high performance aircraft from effectively participating in simulations because these aircraft require a large area of operation and can travel a thousand meters in a matter of seconds. Because of these limitations, and the desire to model a wider range of entities that can participate in simulations, new methods for database and scene management need to be explored.

B. FOCUS

The underlying goal of this thesis is to re-engineer NPSNET to take advantage of the new software and hardware technologies that have been developed in the last few years. By doing this, we improve the rendering rate of the simulation, the realism of the displayed scene, and the extensibility of the system. In addition, we begin the implementation of the Distributed Interactive Simulation (DIS) protocol.

Starting with a beta copy of Performer, a 3-D toolkit made by Silicon Graphic Inc. (SGI), and a modeling tool, MultiGen Flight, the plan was to build a notional terrain database more than ten times the size of the terrain models used previously, and to develop a Performer based simulator to render this new environment. The increased size of the displayed scene enables fast-moving high-performance aircraft to participate in the simulation. Performer's multiprocessing capabilities and database culling allow a much more realistic environment to be rendered. The use of C++ object classes to encapsulate entity behaviors and user input, enhances the systems extensibility.

C. SUMMARY OF CHAPTERS

Chapter II provides an overview of NPSNET-IV. Chapter III discusses the visual database structure and development considerations. Chapter IV gives an overview of the Performer features used by NPSNET-IV. Chapter V provides specific implementation details for NPSNET-IV. Chapter VI provides some C Language Integrated Production System (CLIPS) specific implementation details for the autonomous vehicle controller. Chapter VII discusses the use of the DIS networking protocol. Chapter VIII provides conclusions of the research and further work. Appendix A is a user's guide to NPSNET IV.

II. OVERVIEW OF NPSNET-IV

NPSNET-IV is a virtual world simulator with several stand-alone components integrated together. The first of these components is the Performer based simulator itself which runs on the Indigo Elan or Reality Engine series SGI workstations. The second is a rule-based autonomous vehicle control system which runs in a C Language Integrated Production System (CLIPS) shell and also uses Performer. The third is a spatialized sound system which uses an Emax II digital sound synthesizer controlled by musical instrument digital interface (MIDI) signals from an Indigo workstation, and eight speakers.

Other supporting software includes two C++ class libraries which convert the data flow from the Thrust Master and Kinney Aero flight controls (the two primary input devices for the simulator) into usable input for the simulation code, the DIS network library code [Zesw93] (used for communications between the participating workstations via ethernet), and an environmental effects library [Corb93] (only the smoke generation functions of the library are currently used).

A. MODELING AND GEOMETRY

The current implementation of NPSNET-IV uses MultiGen's Flight format version 12 models to represent the terrain, static features, and dynamic entities. Flight Format is a proprietary visual database format established by Software Systems, San Jose, CA and can be generated using their modeling tool MultiGen Flight [Soft92a]. The Flight format has a hierarchical structure that is easily converted to the run-time database structure for rendering in Performer. A Flight file reader is included with the Performer Development Environment. The database structures of Performer and MultiGen Flight are covered in Chapter III.

Although the goal of this research is not to learn modeling techniques, proper modeling is an important aspect in the design of a database for real-time scene management. A virtual world system must maintain at least ten frames per second to sustain immersion in the simulation [Brys93]. In order to keep the amount of geometry being sent through the graphics pipeline at an acceptable level, the graphic primitives must be developed and stored in a format that allows the simulation software to render all of the geometry that is in the viewing frustum, while rendering the absolute minimum amount of geometry that is not. The process of selecting the geometry to be displayed is called culling.

The precise format of an efficiently structured database is dependant upon the culling method used by the system. It should take less time to cull geometry out of the database than it does to draw it. For example, if the culling takes place at the primitive level in a large simulation database, the system could fail to maintain the desired frame rate. This is due to the computational time required to check each primitive in the entire database to see if it should be rendered in the current scene. On the other hand, if no culling takes place, then every primitive in the database must be drawn for every frame and the graphics pipeline will become the limiting factor. The geometry must be modeled in a way to keep both the culling time and the flow of data in graphics pipeline at acceptable levels.

During the development of the terrain database for NPSNET-IV, we found that the database structure recommended by the *MultiGen Modeler's Guide* [Soft92a] was inappropriate for Performer's culling process [SGI92a]. Figure 1 shows a simple virtual world database containing two sections of ground plane (G1 and G2), two houses (H1 and H2), and two trees (T1 and T2).

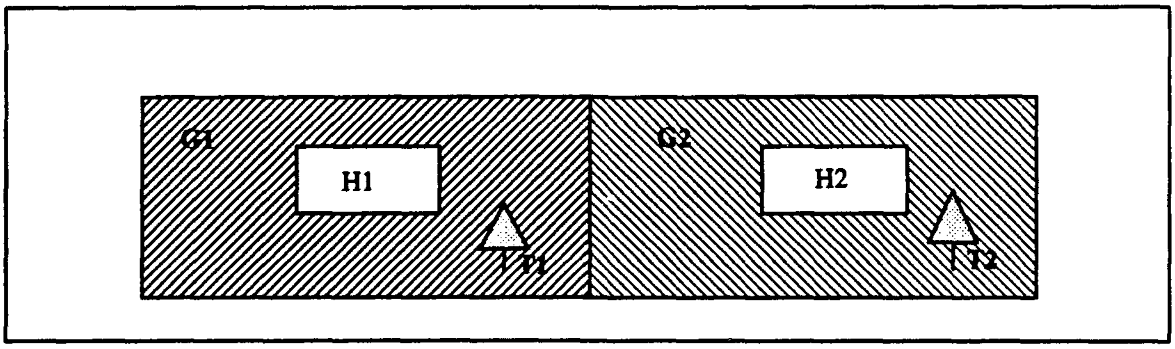

Figure 1 A Simple World Model

Figure 2a shows the recommended MultiGen structure for the example world in Figure 1. This hierarchical structure is logically organized so that all of the objects of one type are grouped together. The problem with this structure is that the bounding boxes that are established at the second level of the hierarchy span almost the entire width of the database. When the cull process checks to see if any portion of these bounding boxes are in the view frustum, there is a high probability that parts of them will be. The cull process will be forced to check for intersection with all bounding boxes on the third level to determine which of those objects are in the scene. As the size of the database increases, these extra intersection tests become computationally prohibitive.

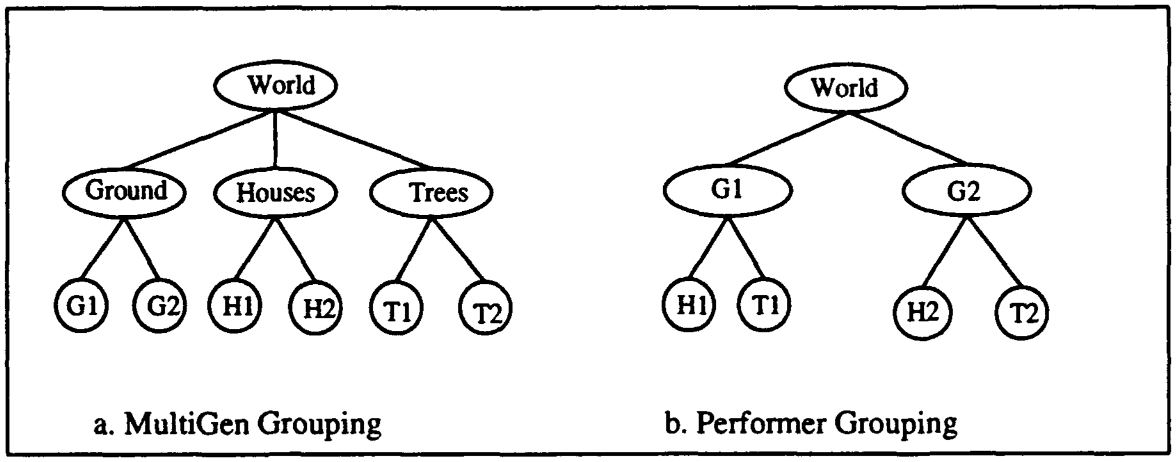

Figure 2 Logical versus Spatial Organization

For real-time applications, Performer's cull process can be optimized by strict adherence to structuring of the database spatially. As shown in Figure 2b, the bounding boxes at the second level of this structure divide the database into two halves. If the bounding box for Ground1 in Figure 1 does not intersect with the view frustum, then none of its children can be in the current scene. This eliminates the need to cull any further down that branch of the hierarchical tree.

B. SIMULATION CODE

The heart of NPSNET-IV is the Performer based simulation code. Performer is a 3D software toolkit developed by Silicon Graphics Inc. (SGI) for developing real-time it to know that it is there. Changes in volume and pitch of the sounds made by the tank provide information on how close and how fast it is moving. When immersed in a virtual world simulation, these aural queues need to be perceived by the participant in the same way they are in the real world. If all of the sounds created in the virtual world are simply piped into a single speaker, or even several speakers, a lot of spatial information is lost.

The NPSNET spatialized sound system (NPSSSS) is designed to make the aural cues more realistic. The system makes all of the sounds in the virtual world sound as if they are emanating from the proper locations relative to a designated entity. The current implementation uses six speakers to produce the spatialized sound, and two subwolfers for low-frequency background sounds such as engine rumble or track noise created by designated entity. For the best effect, the master terminal should be located so that the system's speakers are positioned around it as in Figure 3. Speakers 7 and 8 are subwolfers used for background noise and their placement is not as critical as the other six speakers.

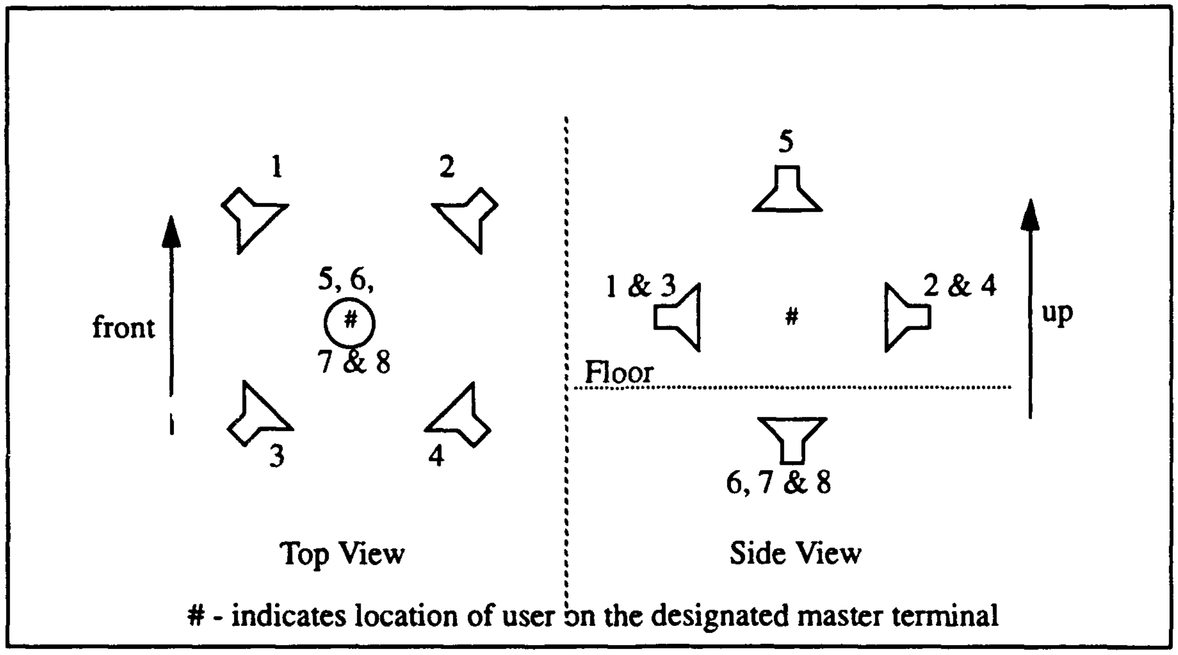
1
2
5, 6,
#
7 & 8
front
3
4
Top View
5
1 & 3
#
2 & 4
up
Floor
6, 7 & 8
Side View
# - indicates location of user on the designated master terminal

Figure 3 NPSSSS Speaker Placement

NPSSSS monitors the network for DIS PDUs from an ongoing simulation. Each time a PDU from the designated master is received, the reference point for determining the positions of all other sounds in the virtual world is updated. As the designated entity approaches certain areas in the virtual world, sound bytes appropriate to that area are channeled to the appropriate speakers at the proper volumes to make it sound as if they are being generated at the correct position relative to the participant. For example, referring to Figure 3, if you fly straight over the top of a waterfall the following sequence will occur. As you approach, the volume of the waterfall's sound will increase from speakers 1 and 2 while maintaining a balance between them. As you get closer, the sound will also increase from speaker 6. As you continue, the volume from speakers 1 and 2 will begin to decrease as speaker 6 continues to increase until you are straight above the waterfall. At this point, the sound will only be coming from speaker 6. Now, as you continue in the same direction the sound will decrease from speaker 6 and increase from speakers 3 and 4. After you have traveled far enough, the sound from speakers 3 and 4 will again decrease and fade out.

By manipulating the volume sent to each speaker, the system is able to make the sound of the waterfall seem to stay at the location of the waterfall in the virtual world. This same principle works with moving objects too. The volume levels are manipulated based on the relative position of the moving object and the designated entity. The heading data from the designated entity is used to maintain the orientation of the speakers so that speaker one is always to the left-front of the master terminal and speaker two is to the right-front.

The feeling of flying through 3D space within NPSNET-IV has been greatly enhanced by the addition of NPSSSS. The 3D aural queues provide such a tremendous amount of information about what is going on around you in the virtual world, it is almost possible to forget that you are looking at a 2D screen.

III. VISUAL DATABASE DESIGN AND DEVELOPMENT

Performer uses a spatially organized hierarchical database structure that is created at run-time for scene management. There is no provision for off-line storage of this visual database, so a conversion program must be used to create the run-time structure from some other database format. The current implementation of NPSNET-IV uses only Flight format models. SGI does provide converters for other database formats and a converter can be written to support any format desired.

There are many similarities in the structure of a Flight database and Performer's run-time structure; however, the current Flight database converter does not correctly handle conversion of every feature. This chapter discusses the similarities and differences in the two database structures.

A. HIERARCHICAL DATABASE STRUCTURE

A hierarchical visual database is a tree-like structure which contains all of the transformations and geometry needed to render a 3D graphics image. In both Flight and Performer, these structures have downward inheritance from parent to sibling for transformation matrices. Most of the structural elements in Flight have an almost equivalent counterpart in Performer which makes the conversion from one to the other a little easier to understand.

1. MultiGen Flight

MultiGen is a modeling tool for creating and editing Flight format visual databases. The structural elements of a Flight database are referred to as beads. There are eight basic types of beads that can be used in a Flight file. A brief description of each is given here to provide a frame of reference, see the *Flight Format Modeler's Guide* [Soft92b] for a complete description of each.

a. Database (DB) Bead

The top-level or root is referred to as the Database (DB) bead. This bead contains the database header information, and is generated automatically when the database is first created. It provides some basic information such as the database units, positional data relative to the earth's coordinates, and the date and time of the last update. The only type of bead that can be attached to the DB bead is a group bead. One or more group beads can be attached to the DB bead. This bead can not be deleted.

b. Group Beads

A group bead is the highest level that can be added to a database. It can contain a transformation matrix which is applied to all of its descendant beads and is the lowest level at which an animated sequence, external reference, or replication can be started Group beads can have other group beads, LOD beads, or object beads attached to them as children.

c. Level-of-detail (LOD) Beads

An LOD bead provides a way of controlling the viewing distance at which its descendants in the visual database will be rendered. LOD beads are useful for controlling the number of polygons that are used in an object's icon. At long distances, an object can be represented by fewer polygons, and as it gets closer to the viewing point, more polygons can be added for detail. Only group and object beads can be attached to an LOD bead.

Each bead contains a switch-in and switch-out distance field which determines the visual range and a center coordinate. If the distance from the eye point to the LOD center falls outside of this range, the node is excluded from the display list as the database is culled. LOD beads can be used in two different ways. The ranges of two LOD beads can be overlapped, or their ranges can follow each other in a sequential manner.

LODs with overlapping ranges, more commonly called additive LODs, allow the basic shape of the object to be put into a low resolution model and each progressive level adds more geometry to the icon. This technique is good for adding such details as door knobs, rear-view mirrors, and window frames to an object as the viewer gets closer. Figure 4 shows an example of additive LODs. The medium resolution house in Figure 4b has no windows, no window frames, and no door frame, and has a total of eleven polygons. To render the high resolution model in Figure 4a, only the thirty-seven polygons that define the window and door frames have to be added by the next LOD node.

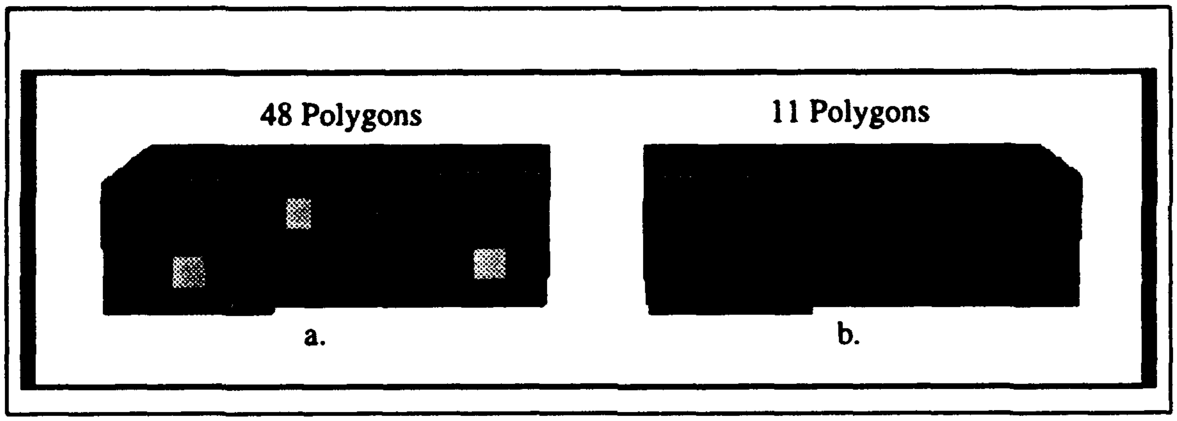

Figure 4 Additive LODs

Sequential ranges are useful for cases where you want to represent a lower resolution model with *geometry that is not a normal part of the higher resolution model*. For example, five polygons are used in the low resolution model of a flat topped house with a recessed roof (Figure 5b), but the higher resolution model (Figure 5a) can not have the vertical polygons from low resolution model because they would show above the roof.

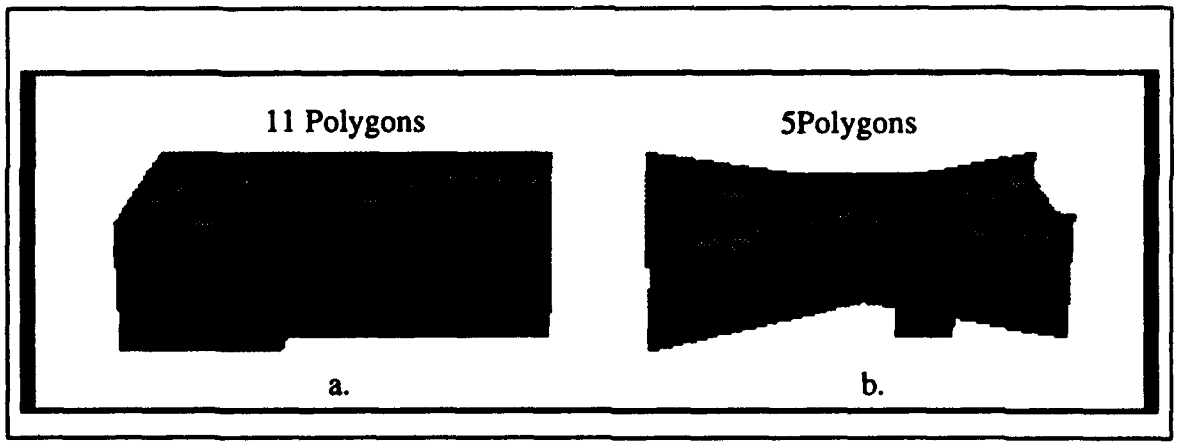

Figure 5 Replacement LODs

d. *External Reference Beads*

External reference beads contain the name of another Flight file that is to be included in the current model hierarchy. When editing a visual database that includes an externally referenced visual database, no modifications can be made to the structure or geometry of the externally referenced model. Attaching the external reference bead to a group bead within the hierarchy allows transformations to be applied so that a Flight file can be used many times in another Flight file.

For example, one model of a tire could be externally referenced four times in the model of a car. Each externally referenced tire would be translated to a different location and two of them would have to be rotated 180 degrees from the other two. For fairly complex models, external referencing reduces the amount of memory needed for storage. Due to the overhead associated with individual Flight files, this reduction or savings is not the case for less complex databases.

e. *Instance Beads*

Instance beads are much like external reference beads, except that they reference a part of the model hierarchy from within the same Flight file. Using the tire example from above, you would construct one of the tires as a part of the car's hierarchy. Then you can make three more instances of the same tire, and apply transformations to get them into the right locations. With instance beads, you can edit the structure or geometry of any of the instance, and the changes will be reflected in all instances. These beads can have group beads or object beads as children.

f. *Object Beads*

Object beads contain a collection of polygons that are somehow related. For example, they can be related because they make up an individual articulated part or consist of one side of a large static model. Objects should be kept as simple as possible to maintain a small bounding box and to increase the flow of the graphics pipeline. Object beads can only be attached to group or LOD beads, and can only have polygon beads as children.

The transparency and shading model to be used on a set of primitives is maintained at the object level. Special render flags for identifying geometry as a shadow and inhibiting its display at day, dusk, or night are also maintained at this level.

g. *Polygon Beads*

Polygon beads contain a collection of ordered, coplanar vertices that describe a graphic primitive which MultiGen refers to as a face. A face can be a point if it has one vertex, a line if it has two vertices, or a polygon if it has three or more vertices. If the polygon bead defines a polygon, then it will also contain the normal for the polygon, texture coordinates, and the color and material for the polygon. Polygon beads can be attached to object beads, or to other polygon beads as a coplanar subface. Polygons should be made convex and planar to ensure the graphics hardware can render them efficiently.

h. *Generic Beads*

A generic bead is used by MultiGen Flight when a node with an unknown opcode is read while converting from another format. Models created using MultiGen Flight will not contain any of these type nodes, so they will rarely occur.

2. Performer

Spatial inheritance is the key to Performers real-time application culling. As the database is constructed, a bounding volume is computed for each node of the tree. As geometry is added to the descendents of a node, that node's bounding volume is adjusted to include the geometry just added. This means that the bounding volume for every child of a given node is contained within the bounding volume of that node. As the database is culled, if a node's bounding volume does not intersect the view frustum, then that node and all of its descendents are eliminated from further culling, and from the draw list. Note: A logically organized database can be converted into a Performer visual database, but the larger the database, the less efficient the application will run, since we loose this spatial inheritance (see Figures 1 and 2).

The basic element in a Performer database is called a pfNode. Unlike MultiGen where each bead type has very specific and different set of attributes, all of Performer's nodes inherit the attributes of a pfNode, which is an abstract data type. The pfNode hierarchy is shown in Figure 6. This inheritance allows Performer to return a pfNode type that can be cast as what ever type node is required. Functions that return a pfNode type usually have the work Node in their name. The attributes of a pfNode include a name, parent list, bounding volume, intersection and traversal masks, callback functions, and data.

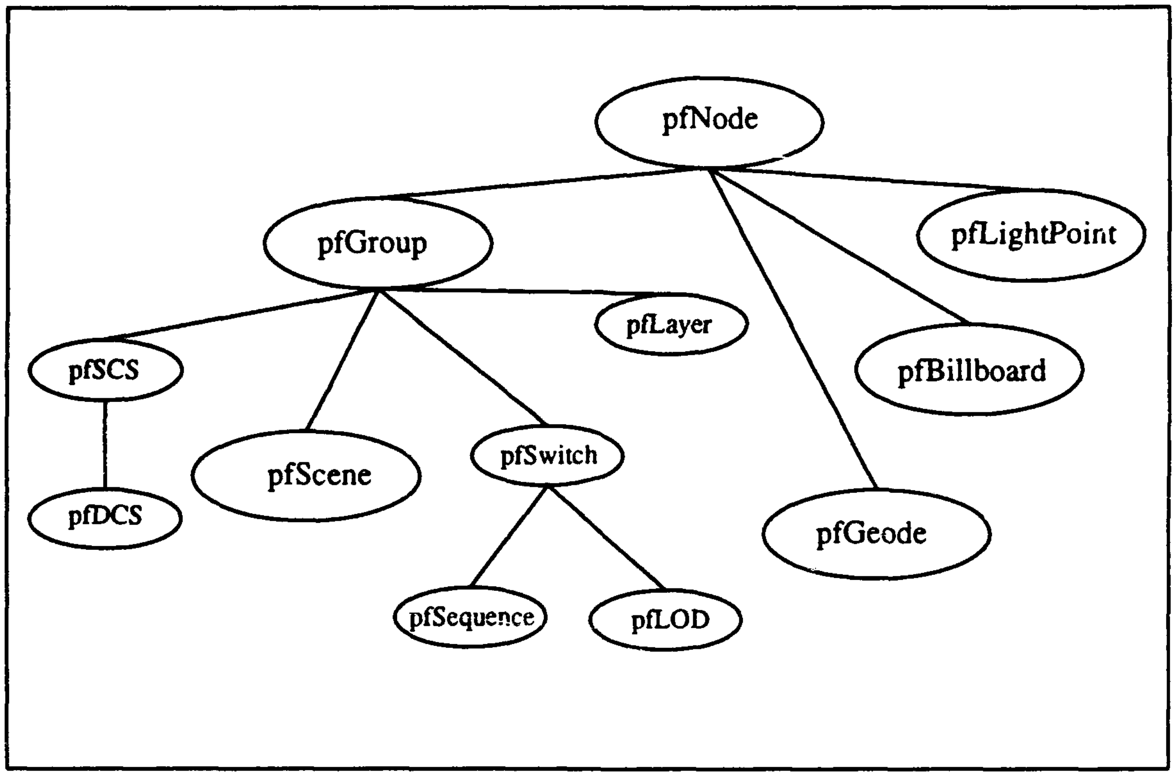

Figure 6 Performer Node Hierarchy

a. pfScene

A pfScene is the root node of the visual database. It is a pfGroup node, and has all of the same attributes with the exception that it can not be attached to another pfGroup.

b. pfGroup

A pfGroup is a branch node in the visual database and can have any other type of node as a child except for the pfScene node. It is used to group other nodes in the database spatially.

c. pfSCS

A pfSCS is a static coordinate system and branch node. Position, orientation, and scale transformations can only be applied when the node is added to the visual database, and these transformations are applied to all of its descendant geometry. This allows instancing of a single model at several locations within the virtual world. After several instances of a model have been transformed and added to the pfScene, pfFlatten can be called, and the transformations will be applied directly to all of the graphic primitives that are descendants of the pfSCS node. Flattening the geometry increases the efficiency of culls and intersections by eliminating the need to do the transformations at run time.

d. pfDCS

A pfDCS is a *dynamic coordinate system* and branch node. This means that in addition to being able to apply an initial transformation to an object, you can continue to modify and apply them at run time. These nodes are used for all object movement, from articulated parts, to actually moving the object around in the virtual world. Most of the transformations that are available in the standard Graphics Library (GL) [SGI91] are provided for the pfDCS, with the exception of being able to scale along only one axis.

e. pfLayer

A pfLayer is a branch node used to enable the proper render of subfaces. A subface is a face that is coplanar with another face, and lies completely within the bounds of the other face which is its parent or superface. Subfaces are used to put windows on buildings, roads on top of terrain polygons, etc.

f. pfSwitch

A pfSwitch is a branch node that can select none, one, or all of its children for display. This type node is good for switching between models for fully operational entities and models for damaged or destroyed entities. The switch is set by the application, and the cull process uses this setting to determine which children to add to the display list.

g. pfLOD

A pfLOD is a branch node that is a special type of pfSwitch node. It is used to control the display of different levels-of-detail child nodes based on the node's distance from the view point. This selection process occurs in the cull process. The functioning of Performer's level-of-detail nodes is the same as the LOD beads in MultiGen Flight. See "Level-of-detail (LOD) Beads" on page 12.

h. pfSequence

A pfSequence node is a branch node that is also a special type of pfSwitch node. It is used to create an animation sequence in a Performer application. The animation sequences through the child nodes in a cyclic or swing fashion. The display time for each child can be individually set, the total duration of the sequence can be set, and the sequence can be turned on and off at run-time. The application process sets the required values, and turns the sequences on and off. The cull process determines which nodes to add to the display list based on the parameters provided.

i. pfGeode

A pfGeode is a leaf node that contains a list of geometry. The geometry is contained in pfGeoSets, which also contain state information for the geometry.

j. pfBillboard

A pfBillboard is a pfGeode that always faces the view point. As the view point moves, the billboard pivots, but does not translate. A common use of a billboard is for trees, which can be modeled as a single textured polygon [Soft92c].

k. pfLightPoint

A pfLightPoint is a leaf node that represents a point light, or a set of point lights. An individual light point or set can be given a direction, shape, intensity and color.

B. FLIGHT TO PERFORMER CONVERSION

Performer provides a procedure that reads and converts a Flight format model into a Performer run-time visual database structure. This procedure returns a pointer to a pfGroup node, which means it can be attached to a pfGroup, pfSCS, pfDCS, pfLOD, pfSequence, or to the pfScene directly. The converter recursively traverses the Flight file structure, and converts the Flight database beads into an appropriate Performer node.

Some of the Flight database beads convert directly to Performer nodes with the same types of attributes. For example, an LOD bead converts directly to a pfLOD node. Other beads can convert to several different types of Performer nodes, for example a Group bead can convert to a pfGroup, a pfSCS, a pfDCS, etc.

The Flight file converter does not properly convert all Flight database beads and their attributes to the proper Performer counterparts. A group bead that is the parent of an animated sequence is not converted to a pfSequence node. When a Flight animated sequence is loaded and then rendered, all of its children are rendered in every frame. All of the animation information is lost.

Light points and strings are another problem area. MultiGen allows them to be constructed as objects, and the light points do not have to be coplanar. However, the light points will only transform correctly to Performer if all of the light points in a single object are coplanar.

Externally referenced files in a Flight file are read in and the proper transformations are applied, but the loader gives a warning message that an unknown transformation matrix was found. Although annoying, this warning does not appear to be a real problem.

C. BUILDING AN ANIMATED SEQUENCE

Animation sequences can be used in a variety of ways in a simulation system. The turning of a helicopter's blades, spinning of a vehicle's wheels, and the flashing of fire during an explosion can all be animated using an animated sequence. Since the Flight format converter does not properly convert animated sequences, the simulation code must construct any animation sequences it uses during its initialization.

The first step in building an animated sequence is to create a pfSequence node. Then each frame for the animation sequence is constructed and attached as a child node. Each frame could be a complete new Flight file read into a pfGroup, or could be a scaled, rotated, or translated copy of another frame. For instance, to animate the rotation of a vehicle's wheel would only require one Flight model. Each child of the animation sequence would have an instance of the wheel with a different rotation applied to it.

After the nodes for the animated sequence have been added to the pfSequence node, several sequence parameters must be set. The first is the sequence time for each child. The sequence time determines how long each child is displayed and can vary from child to child. Differing child sequence times allows for implementing animation techniques like slow-in and slow-out with fewer child nodes in the sequence. The second parameter is the sequence interval which sets the range of the children to be displayed in a sequence. The third is the sequence duration which determines the speed and number of times that the animation will occur.

The final parameter is the sequence mode. There are three states for a sequence: running, pause, or stopped. The difference between paused and stopped is where the sequence can resume. If the sequence stopped, it will start with the beginning child node. If the sequence is paused, it can be resumed starting at the child where it left off.

All of the parameters for a sequence can be modified at run time, which means you can control the speed of the wheel rotations as the speed of the vehicle changes. You can also get the current frame for an animation, which could be used to tell what color the animated traffic lights at an intersection are as you approach.

LOD nodes in Performer can be built in the same way as described here for animated sequences, loading a different Flight file for each level of detail. Even though the Flight converter does handle these conversions properly, building the LODs within the application would provide a means for changing the child node ranges at run-time. This could be used to help control system load, by extending the ranges when the frame rate decreases.

IV. PERFORMER OVERVIEW

NPSNET-IV is a Performer based, real-time, interactive simulation system. Performer handles many programming details that make real-time virtual world application development much easier. This chapter details the major Performer features that the NPSNET-IV simulation system currently uses. These include multiprocessing support, visual simulation routines, mathematics support routines, and intersection testing. See the IRIS Performer Programming Guide [SGI92a] for a complete description of Performer's features. Some of the other features used by NPSNET-IV are underlying routines that are almost transparent even to the application developer and are either not discussed, or are discussed with closely related material in other parts of this thesis.

A. MULTIPROCESSING AND SHARED DATA MANAGEMENT

One of the major features of Performer is its multiprocessing management. Although Performer does handle many of the multiprocessor configuration and management requirements, the application must be developed with the proper shared memory arenas to support multiprocessing.

1. Performer Processes

Performer configures the application code to run in a pipelined multiprocessing model using up to three separate processes. The number of processes it creates is based on the number of processors available on the host workstation and the number requested. When three or more processors are available and requested, Performer will create separate application, cull, and draw processes. Figure 7 diagrams the Performer environment for NPSNET-IV when three processors are available and requested.

In this three process mode, a two frame latency is introduced into the system because as the draw process is drawing frame N, the cull process is working on frame N+1, and the application process is working on N+2. As the user reacts to the displayed scene,

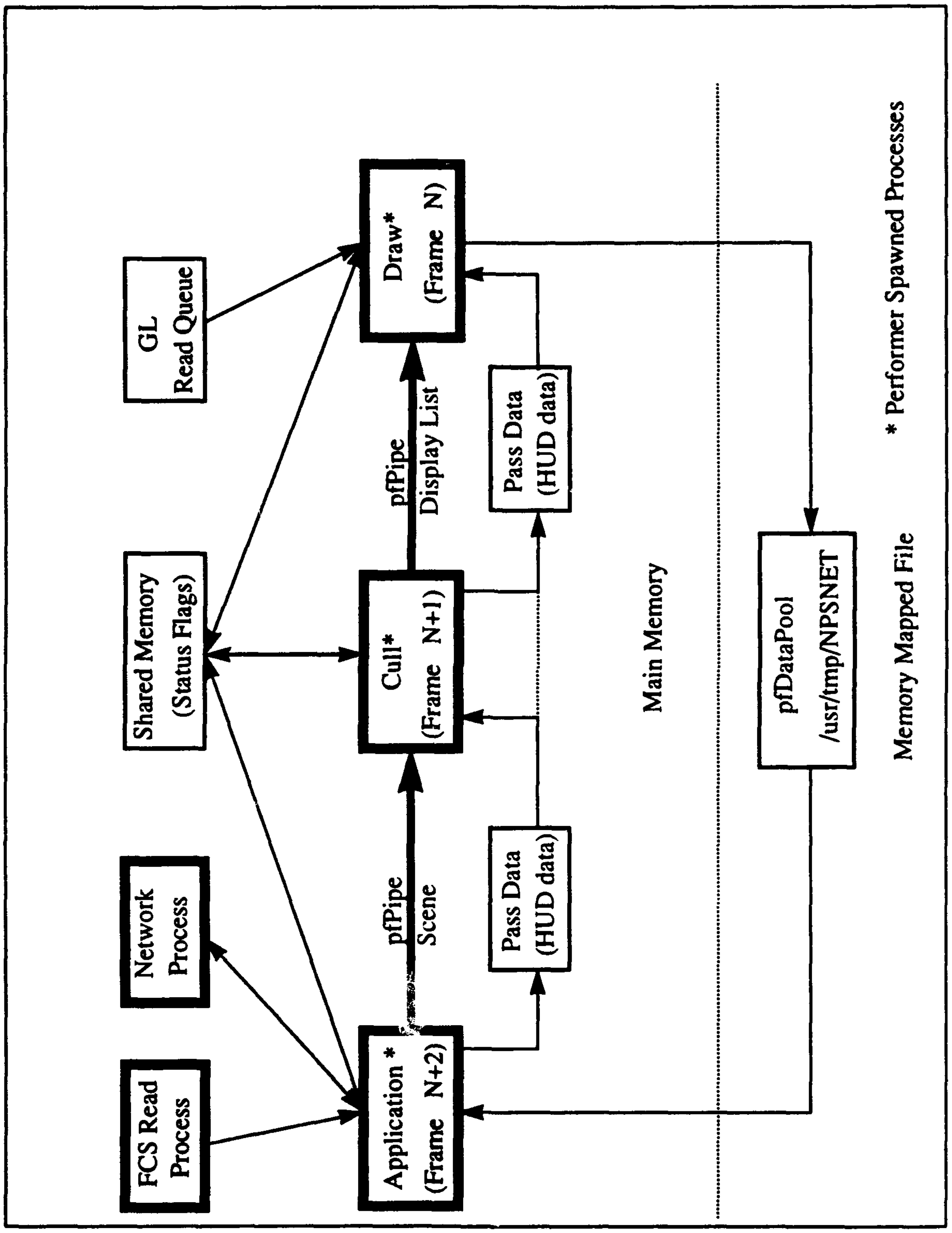

Figure 7 Processes and Data Flow in the Performer Environment

this new input information is not available to the frames that are being processed by the cull and draw processes, so the response is not rendered until two frames later.

Figure 8 shows that this two frame latency with a three process model still provides an improvement over the latency that a single processor model delivers because of the reduced time per frame and increased frequency of the check queue (CQ) in the application process [Henn90]. In Figure 8, event A is read from the event queue by both the single and triple threaded applications during application process time for frame zero. Event B, occurs just after the check queue for frame zero, so it is read by both applications during the application process time for frame one. The next check event queue for the triple threaded process occurs during the cull time for frame zero in both applications. The single threaded application does not get to the next check event queue until after the triple threaded application is already drawing frame one, with event B already processed.

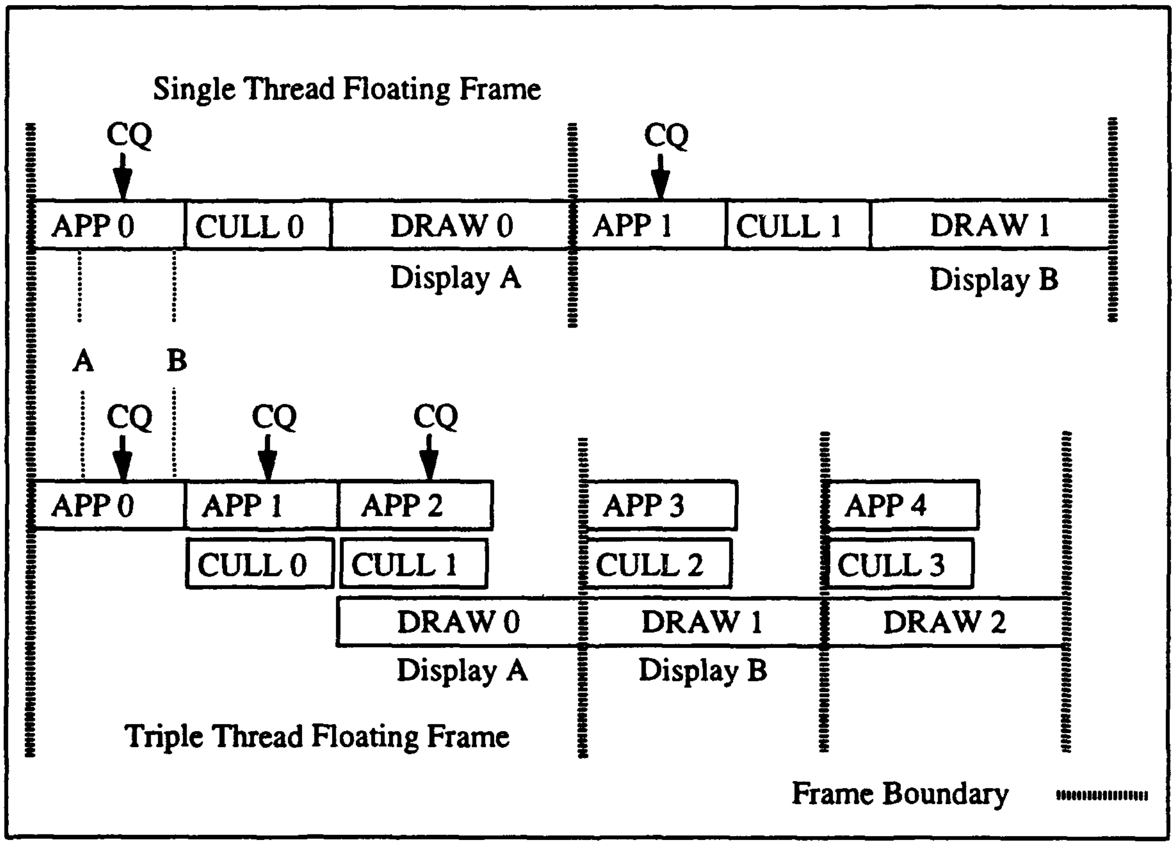

Figure 8 Single and Triple Thread Latency

The goal of using a multiprocessor model is to keep the frame rate at an acceptable level. Performer allows you to set the desired frame rate for an application, but it must be a factor of the monitor's video refresh rate. If that rate or higher can be achieved, then Performer maintains the requested rate. For example, if we ask for 60 frames per second (fps), when the graphics and application loads are low, Performer will deliver 60 fps. As the graphics or application load increases, if 60 fps can not be delivered, then Performer will reduce the frame-rate to 30 fps, then 20 fps, then 15 fps, etc. as required to allow for all three processes to finish before the start of the next frame. These factors of the video refresh establish what are referred to as frame boundaries, and are used to keep the Performer processes in sync with each other.

The frame-rate can be allowed to float, or to be set at a fixed value. If it is allowed to float, then the three processes begin work on a new frame as soon as one of the frame boundaries is reached. This means that the number of frames drawn in a given amount of time can vary, as the scene complexity varies. If a fixed frame-rate is used, the same number of frames will be drawn in a given amount of time, so frame dropping (skip the rendering of one or more frames) is used to maintain the rate as the scene complexity increases. [SGI92a]

a. Application Process

The application process does a majority of the initialization and also contains the main graphics simulation loop. During initialization, all the shared memory arenas required for multiprocessor operation are established, and any cull and draw callback functions are declared. Each time through the simulation loop, the application process is responsible for calculating the position of the current view frustum, the positions of all entities in the simulation, and updating the visual database to reflect these changes.

After the multiprocessing model for Performer has been established, the Performer application process can create other processes. These child processes can not make Performer graphics calls, but are a part of the shared process group and can access the same virtual memory space.

The main graphics or simulation loop in a Performer application has a fairly simple five step structure.

- A call to pfSync() which puts the processes to sleep until the next frame boundary is reached.
- Perform latency critical tasks such as positioning of the view point.
- A call to pfPassData, if there is data that needs to be passed to the cull and/or draw processes.
- A call to pfFrame to initiate the cull traversal and draw for the newly processed frame.
- Perform time-consuming calculations and update the visual database for the next frame.

When pfFrame is called, a copy of the visual database is made and passed to the cull process for further processing via the pfPipe. If pfPassData is called, a copy of the user defined pass-through data is made and passed on to the cull process. For a graphical representation, refer to Figure 7.

b. Cull Process

The cull process traverses the scene and builds a display list of geometry that is passed to the draw process. The selection criteria for the geometry that goes onto the display list is based upon the position of the view frustum and the parameter values found in pfSwitch type nodes. If a node is turned off by a pfSwitch or pfSequence node, then that node and all of its descendants are excluded from the display list.

For most nodes, their bounding volume is checked for intersection with the view volume. If there is an intersection, the cull process continues down the tree, checking the bounding volume for each child node for intersection with the view volume. When no intersection occurs, the cull process terminates its traversal down the current branch, and the current node and all of its descendants are not added to the display list.

In addition to the intersection testing with the bounding volume and the settings of pfSwitch and pfSequence nodes, the cull process must check pfLOD nodes. The cull process uses the assigned display ranges for each of the child nodes to determine which, if any, should be added to the display list.

Stress parameters are another selection feature provided by Performer. If stress parameters are used, then lower LODs could be substituted when the system load reaches a critical point. The normal display ranges are used when the system loading is not at a critical point.

Additional culling criteria can be implemented using a cull callback function. If a callback is defined, it is automatically invoked each time pfFrame() is called in the simulation loop, and it must include a pfCull() function call to initiate the standard Performer cull process. When the cull process completes a frame, it passes the display list for the new frame to the draw process via the pfPipe. If pass-through data is being used, it too is passed to the draw routine.

c. *Draw Process*

The draw process renders the geometry passed to it in the display list, and can access the passdata and the flags in shared memory when required. The draw process owns all of the workstation hardware as far as input and output operations are concerned. This means that the draw process must perform the tasks of reading keyboard, mouse, and Spaceball inputs.

A draw callback must be used to implement any additional drawing requirements such as a graphical user interface (GUI) or overlay mask for the scene. All defined draw callbacks are automatically invoked at the conclusion of the cull process. If a draw callback is defined, a pfDraw() function call can be made to initiate the normal Performer draw process if desired.

Several pfDraw() calls can be made in a draw callback to do multi-pass rendering using accumulation buffer techniques. The stereo graphics implemented into

NPSNET-IV also use a double call to pfDraw(), but uses the stereo hardware feature rather than the accumulation buffer.

The draw process passes user input information back to the application process using a pfDataPool and a shared memory arena. Any user input that results in a hardware configuration change can be handled by the draw routine; all of the other inputs must be returned to and handled by the application process. Time critical user inputs, like vehicle control inputs from the keyboard, are passed back to the application using the pfDataPool. Other user inputs, like switching input control devices, are not time critical or frame dependent. These inputs can be signaled to the application using a flag in the shared memory arena.

2. Process Data Communication

The key to successfully multiprocessing a Performer application is a thorough understanding of the processes created and how these processes communicate with each other. Because each of the processes can be working on a different frame, most of the data flows in the Performer environment are one-way. Figure 7 shows the direction that data flows between the processes in a multi-processor environment, and the type of shared memory arenas used to pass the data.

a. Pass-through Data

Pass-through data is a one-way path for data from the application process to the cull and then draw processes. This arena must be kept as small as possible because copies for three consecutive frames exist at any given time, and there is a performance penalty associated with passing this data [SGI92a].

This arena is used to pass data to user defined cull and draw callback functions that require frame-dependent application data to modify the standard Performer cull and draw applications. Pass data is not required if the callback functions are created just to handle user input.

b. Shared Data

In most multiprocessor environments, the processes can communicate with each other through a shared memory arena that is created before the sproc or fork call is made. The shared data arena in the Performer environment can be accessed by all of the processes that the system spawns. Because of the two frame latency present in the environment, no frame-dependent data can be passed using the shared memory arena. No semaphore is used for this arena. The application accesses this data at the end of the simulation loop, and the draw process accesses at the beginning. The pfSync and pfFrame calls ensure that the access times can not overlap.

c. pfDataPool

Another type of shared memory arena used by Performer is a data pool, which is a memory mapped file. A data pool lock that must be used to guarantee mutual exclusion is maintained for this arena. Any number of pfDataPool arenas can be created for passing data between the three Performer processes.

One use for this type of arena is to pass frame dependent user inputs from the draw process to the application process. The draw process reads the GL queue, sets the data pool lock, writes all inputs that require processing by the application to the file, and then resets the data pool lock. The application process sets the data pool lock if the arena is available, reads and processes the data, and resets the data pool lock.

B. VISUAL EFFECTS SUPPORT

Visual effects can greatly improve the realism of virtual world simulations. Performer provides some functions that simulate environmental, and atmospheric effects that make adding these effects to a simulation very easy.

1. Earth and Sky Model

The pfEarthSky is a set of functions that perform the screen clear function and also provide effects such as fog and clouds. This function is automatically called if no draw callback is defined. This function will allow you clear the screen to all one color, clear it so that only the sky and horizon are drawn, or clear it so that the sky, horizon, and earth are drawn.

With the earth enabled, you can use a model with a smaller ground plane than you could without it because Performer provides a ground plane. Performer's ground plane attributes can be set to match the models. As the edge of the model is approached, the ground plane will continue to the horizon, rather than those areas of the screen being cleared to a background color.

The earth and sky share a common color at the horizon. This common color allows the sky to be blended so that there is a smooth transition with the ground plane at the horizon. The width of the blend ban is defined in degrees by the application. The colors used in the earth and sky model can be changed at run-time, which allows the application to simulate changes in the time of day.

a. Fog

Performer provides two types of fog; general fog, and ground fog. The density of the general fog is uniform throughout the display and provides the general visibility level for the scene. The ground fog starts at a user defined altitude, and transitions from the general visibility level at the top, to the user selected visibility level at the ground.

b. Clouds

The earth and sky model also provides for a layer of clouds to be simulated. The cloud layer is an opaque region that is set by an upper and lower altitude. The color at the top and bottom of the layer can be individually set, and the layer is blended smoothly from one color to the other. The altitude levels and colors of the clouds can also be change during run-time.

A blending layer is also provided above and below the cloud layer. The vertical thickness of these two layers is set individually. These layers provide for a smooth transition from the cloud layer to the general visibility areas above and below it.

2. Light Points

Runway lights, traffic lights, etc. can be simulated using a pfLightPoint. The size of the light point is dependent upon the workstation's hardware and the graphics display mode used. Performer uses GL anti-aliased points to represent lights, when this capability is available. The color, location, and orientation of light points can be changed at run-time, which allows a single light point to act as a rotating beacon, or as a string of strobe lights.

C. MATHEMATICS SUPPORT ROUTINES

Performer provides a high-performance interface to the standard IRIS graphics library that extends the functionality of some of the GL routines and also provides some utilities that take advantage of the new hardware features available on the Reality Engine workstations. Two areas that Performer provides significant support for are vector and matrix manipulation. Refer to the *IRIS Performer Programming Guide [SGI92a]* and *IRIS Performer Man Pages* [SGI92b] for a complete list and description of these operators.

1. Vectors

Performer defines a three component vector type as a pfVec3, and a two component type as pfVec2. These two types can be used to represent actual points (a vector from the origin), or a normalized vector. The same functions are provided for both, with the exception of two transformation functions that only work with the three component vectors.

Functions to manipulate a single vector, or to perform mathematical operations on two vectors are provided. Some of the manipulation functions are copy, negate, scale, normalize, and find the length of a vector. Some of the mathematical operation functions are add, subtract, scale, combine, and test the equality of two vectors.

The two transformation functions for three component vectors proved to be invaluable in the development of NPSNET-IV. These two functions are used to transform a vector or point using a 4 x 4 matrix. Because Performer uses a different entity coordinate system, calculating a bearing to target and even finding the velocity vector using trigonometric functions is tedious. These two functions take all the work out of calculating these two critical items.

2. 4 x 4 Matrices

Matrix manipulation and mathematical operation functions are provided for 4 x 4 matrices. In addition to the standard matrix operations, Performer provides the capability to build up a matrix stack. These stacks can be manipulated using the Performer commands the same way the IRIS system stacks are.

One of the commands that proved very useful was the pfMakeEulerMat, which takes the heading, pitch, and roll angles as input, and returns a 4 x 4 matrix that can be used to find the entity velocity vector as described above.

3. Volumes

Performer provides five structures that are used as bounding volumes, spheres, axially-aligned boxes, cylinders, half-spaces (one side of a plane), and frustums. Functions are provided that can be used to build custom bounding boxes for culling purposes, but the Flight format loader automatically calculates the default type bounding volumes for all nodes. These bounding volumes can also be used for intersections testing.

D. INTERSECTION TESTING

One feature that every virtual world needs is collision detection. If you can fly through buildings and other vehicles, the simulation is not believable. Performer provides a wide variety of intersection testing routines that can be used for collision detection and also for entity control routines.

Intersections can be done between a line segment and the bounding volume of any node, between a line segment and the actual geometric primitives, between a point and a volume, and even between two volumes. The routines provided for creating a volume can be used at run-time to build a bounding box that encloses the area an entity will occupy in the next frame. This volume can then be tested for intersection with the bounding volumes of other entities and static features in the virtual world. Volume intersections are the fastest, but since the bounding volumes are simple geometric shapes, an intersection could be detected when there was no geometry that actually intersected.

To facilitate intersection testing, Performer uses intersection masks. The terrain in a database could be assigned one mask value and water another. This difference in the masks would allow the simulation code to have ground vehicles stay on top of the terrain, but sink in through the water. An object can have several masks set by using a logical or to combine the desired masks. For instance a vehicle could have a vehicle mask, a truck mask, and a force mask.

Intersection masks can be set during run-time. There are two different modes that an intersection mask can use, cached and uncached. The cached mode is supposed to be used for static features in the world, and the uncached for dynamic. We had a problem with using the cached mode after switching to the Flight Version 12 loader. This problem has not been resolved at this time, so all of the masks used in NPSNET-IV use uncached masks. When we attempted to create a mask with cached normals, the code would crash on the Performer call that creates the mask.

V. IMPLEMENTATION

A. C++ OBJECT ORIENTED PARADIGM

One of the most important requirements for the development of NPSNET-IV was that the system be extensible. Due to the transient nature of most of the NPSNET researchers, it is not only important that the system can be extended, but that the extensions can be made with as little modification to existing code as possible. Each time the original code is modified provides an opportunity to introduce problems into the system. With time, the time spent correcting induced errors is greater than the time spent trying to correct programming errors in the original code [Broo75].

The object oriented paradigm is ideal for this type of incremental development because it allows researchers to build on the already proven behaviors of existing objects. The two areas that NPSNET-IV takes advantage of the object oriented paradigm are for entity behaviors and user interface device drivers.

1. Entities

Each entity in an NPSNET-IV simulation is created as an object derived from the VEHICLE or MUNITION classes. Each object is responsible for updating and maintaining its own orientation data and status in the simulation. The driven VEHICLE object for each simulator must also process the participant's inputs, broadcast its orientation data and status to the network, and react appropriately to the control data received. Each MUNITION object created by the simulation must also broadcast its orientation data and status to the network. Figure 9 shows the MUNITION class hierarchy, and Figure 10 shows the VEHICLE class hierarchy.

The only MUNITION objects that can be present in a simulator are the ones that are created locally. A global array is also used to manage these objects. When a munition is fired or dropped by the driven VEHICLE, a MUNITION object is created and added to the G_missile array. The new MUNITION object follows the same rules for sending out

PDUs as the VEHICLE objects. When it sends out an entity state PDU, all of the other simulators add the entity to their G_vehlist array, and treat it as a VEHICLE object.

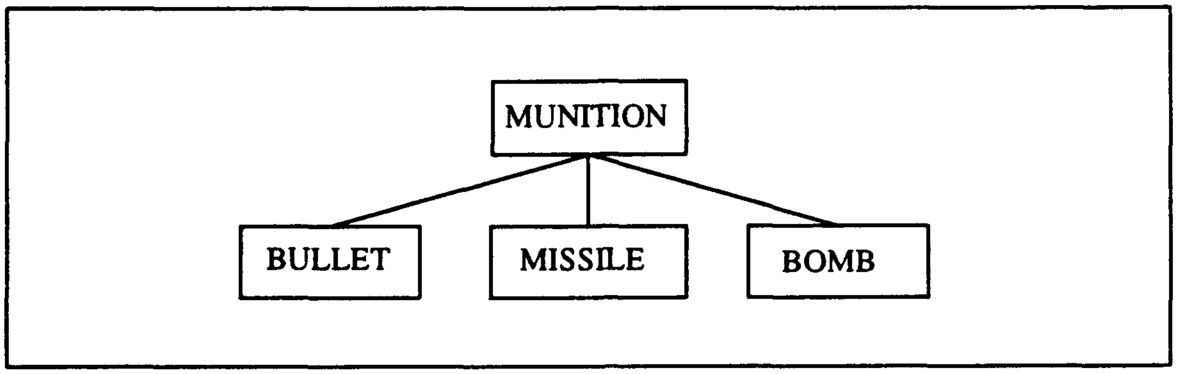

Figure 9 MUNITION Class Hierarchy

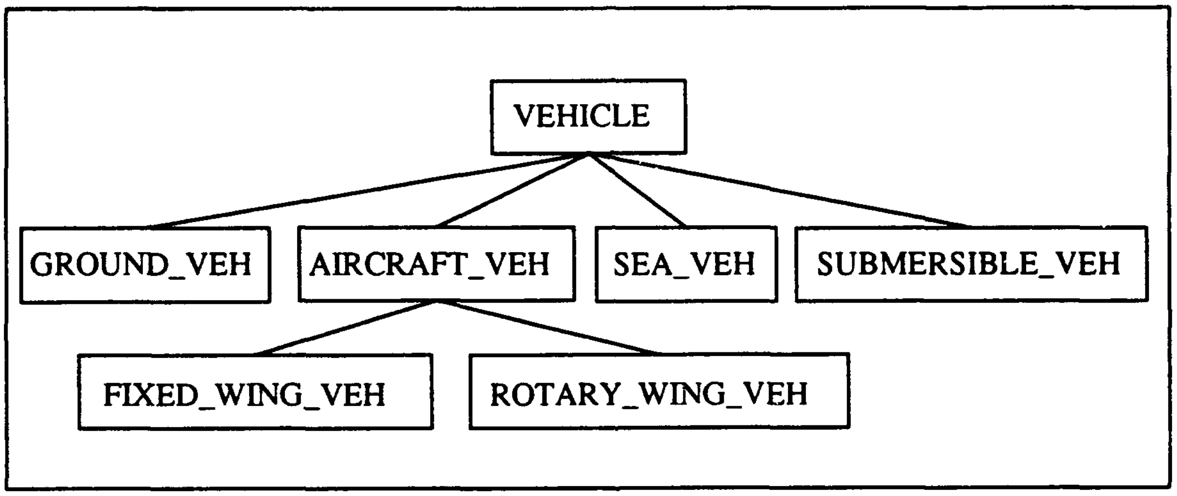

Figure 10 VEHICLE Class Hierarchy

The object oriented paradigm used to develop NPSNET-IV also provides for rapid addition of new entity types in the simulation system through inheritance of the behaviors defined for existing entity types. After the initial development and testing of NPSNET-IV, an autonomous vehicle control system was developed to help populate the simulation. One of the autonomous agents was a Loch Ness monster that would chase low flying aircraft.

After initially casting this entity as a sea vessel, and then as a base vehicle type, it was determined that a new entity type was needed. As a sea vessel, the Loch Ness monster was restricted to the surface of the lake. As a base vehicle type, the remote entity positioning algorithm used to move the entity's representation in other networked simulators would cause the monster to rise up out of the water instead of just coming to the surface. The Loch Ness monster needed the same kind of movement restrictions that a submarine or fish would need. It needs to move in and be restricted to the water, but does not have to remain on the surface. A new entity type, submersible, was added to the simulation system in less tha thirty minutes.

2. Controls

The flight control and headmounted display interfaces were also developed as C++ objects. The first flight controls that were implemented were the Thrust Masters. The Thrust Master interface took several weeks to develop and debug so that proper integration into the simulation code could be achieved. The development of the interface for the Kinney Aero flight controls took just a few hours, and provides an almost identical set of interface functions for the application program.

B. CONFIGURATION FILES

NPSNET-IV makes extensive use of configuration files. The system executable "npsnetIV" can take any number of command line arguments. Each command line argument is treated as a configuration file, and the last value loaded for each configuration option is what is used when the system actually starts. The system will start-up and run using default values if no configuration files are present.

Several of the configuration options involve datafiles, which are used to control options like what models are loaded into the global vehicle types array, what terrain models are loaded, and identification of workstations on the local network. For example, one of the configuration options is the TERRAIN_FILE. The TERRAIN_FILE is used to load all of the Flight models that require intersection masks for collision detection and entity control. Appendix A gives a complete description of all of the configuration options.

C. INTERFACE OPTIONS

NPSNET-IV provides several interface options that have not been available on previous versions. One of the most significant is the use of stereo graphics, using CrystalEyes shutter glasses. Stereo greatly enhances the feeling of 3-D flight by adding another dimension to the display screen.

The second option is that the size and format of the display can be configured to different formats. Depending on the size selected, different stereo and anti-aliasing modes can be used. The Reality Engine workstations allow up to sixteen multisamples for antialiasing, if the proper size window for the hardware configuration is used. To do a full screen display with sixteen multisamples requires four Raster Managers. All three of our Reality Engines currently have only one Raster Manager, which limits the full screen display with all geometry being culled out of the scene to twenty frames per second. This limitation is due to pixel fill.

The user also has the choice of four different input devices. The Thrust Master flight stick and throttle, the Kinney Aero flight stick and throttle, the Spaceball and the keyboard. The two flight stick and throttle options are interfaced to NPSNET-IV through C++ objects. If muliprocessing is available and requested, the flight control interface objects are spawned as a separate processes. These processes are suspended when their device is not actively controlling the simulator.

NPSNET-IV uses a heads up display (HUD) that is drawn on top of the current view. Figure 11 shows the HUD with all options enabled. The default mode has the aim point, horizon indicator, other participants, and status information (the text on the left and right sides) displayed. The 1, 2, 3, F1, F2, F3, and F4 keys are used to control what information is displayed on the HUD. See "NPSNET IV USER'S GUIDE" on page 58.

The F4 key cycles through six different modes, these modes let the user select which symbols are displayed. You can have no symbols, then add the aim point and horizon indicator, then add the other participant icons, then add the range rings, then add the landmarks, and finally add in the world boundary.

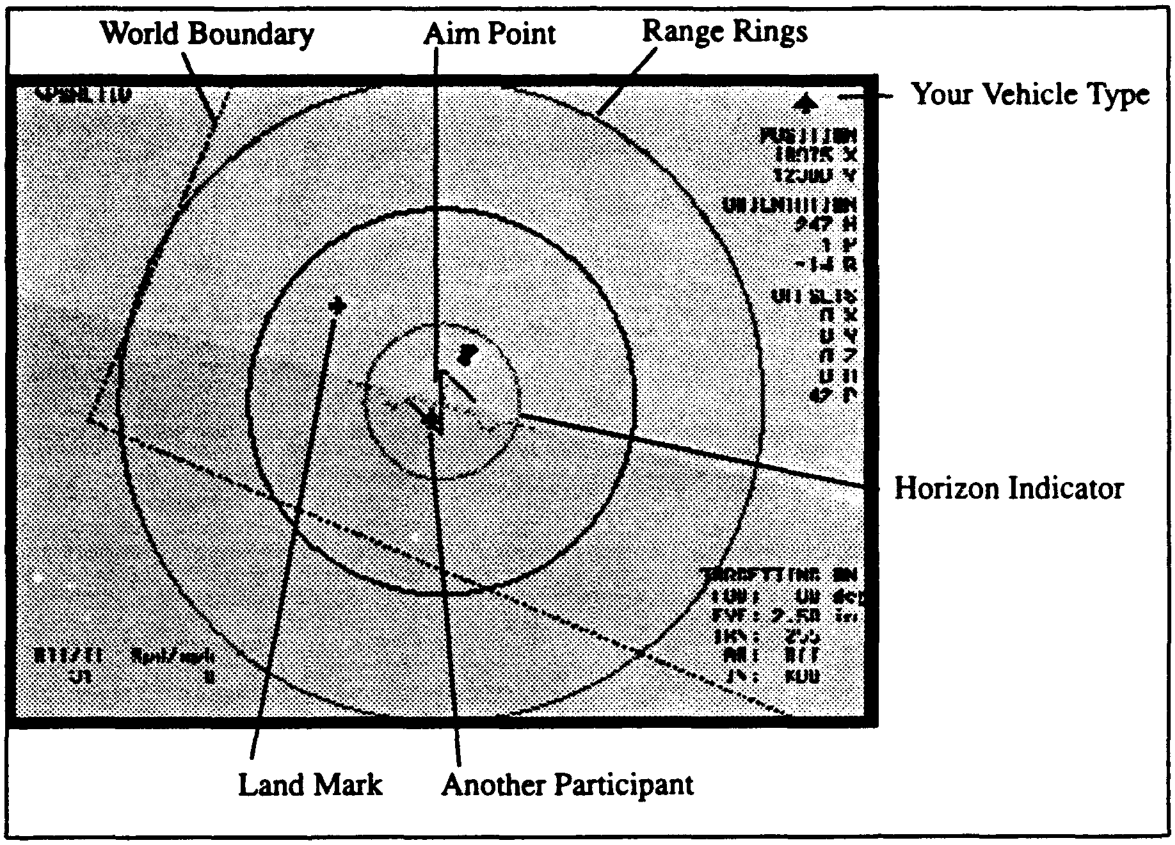

Figure 11 NPSNET-IV Heads Up Display (HUD)

D. DATA STRUCTURES

1. Pass-through Data

The pass-through data in NPSNET-IV is used to draw the heads up display (HUD) information. A single structure called PASS_DATA was used to pass the required information to the draw process. Figure 12b shows this structure, which contains the driven vehicle's posture, speed, type, view offsets, and an array of the closest twenty-five entities.

Figure 12a shows the structure used for the hud_data array in PASS_DATA. This structure includes the information required to properly encode the HUD data with the entity type, force, and elevation. Its location is used for proper placement on the HUD, and to determine which vehicle is being selected when the tether option is used.

```
typedef struct
{
    pfVec2 location;
    float elevation;
    unsigned char force;
    int type;
    int vid;
} HUD_VEH_REC;
```

a.

```
typedef struct
{
    pfCoord posture;
    pfCoord look;
    int type;
    float speed;
    HUD_VEH_REC hud_data[HUD_MAX_VEH];
}PASS_DATA;
```

b.

Figure 12 NPSNET-IV PASS_DATA

2. Shared Data

Shared data in NPSNET-IV is used mainly for control flags, but some data that is not effected by latency is also passed to and from the application through this arena. The data that is passed to the draw process includes the clipping plane data, the current view port size and origin, the world bounding box, and a pointer for the down loading of textures during the first call to draw. The data that is passed back to the application includes the mouse location, and the vehicle identification for a vehicle when it is selected for pursuit or to be tethered to.

The shared data values are set by a combination of values provided by the user supplied configuration files, and by a hardware inventory that is done during initialization. This inventory determines the level of graphics support available, i.e. Reality Graphics or Elan Graphics, the number of multi-samples available for anti-aliasing, and what stereo mode can be supported if any.

3. Global Data

NPSNET-IV uses nineteen global variables. Global variables are accessible to all of the functions and processes in NPSNET-IV, but care must be taken in how these variables are used. For instance, the scene graph should never be modified by the cull and draw processes because of the frame latency created by multi-processing.

Three of the global variables are associated with the arenas used to pass information between the draw and application processes. Shared data is one of them, and the memory mapped file is another.

There are five pfGroup pointers, one for each major sub-branch of the scene. These branches are

- G_dirt - Contains all of the terrain, water, and foliage.
- G_veh - Contains geometry for all current vehicles from the network.
- G_obj - Contains all of the static object geometry.
- G_missiles - Contains the geometry for MUNITIONS fired by the local entity.
- G_flame - Contains the geometry for the smoke columns and fireballs.

There are four arrays used to manage the entities within the world. These four arrays are:

- G_vehlist - Contains the driven vehicle information and information on all entities from the network.
- G_vehtype - Contains the name and pointer to the geometry associated with each type of dynamic entity that the simulator can use.
- G_objtype - Contains the name and pointer to the geometry associated with each type of static models used in the virtual world.
- G_weaplist - Contains the data for MUNITIONS fired by the local entity

There are also some float types that are used to control the weapons fire rate and to hold the current time as established at the beginning of each loop.

E. CALL BACKS

Function callbacks can be used for custom culling, drawing, and intersection testing. NPSNET-IV uses a draw callback function for drawing the heads up display, and a pre-traversal callback function for collision detection.

1. Drawing Routine

NPSNET-IV makes extensive use of keyboard and Spaceball inputs, which require the use of a draw call back to pass this input data to the application. In addition, the standard Performer draw routine has no provisions for drawing an overlay such as the HUD. It is only passed the draw list containing the scene geometry. In order to draw the HUD, a custom draw routine must be performed.

The draw callback reads the GL queue, which is used for input from the keyboard and Spaceball [SGI91]. Those inputs that deal with changing the HUD display, the monitor mode, and other hardware related functions are processed by the draw routine as it reads the queue data. Input that must be passed to the application, such as the driven vehicle control inputs, are written to a pfDataPool for processing during the next frame time.

The draw routine also handles the stereo graphics capabilities of the system. When stereo is requested, it determines if the current configuration can support stereo, and if it can, changes its drawing mode to provide stereo. On a Reality Engine workstation, there are several different modes that support stereo and these require two completely different drawing methods to support them. Flags and data values in the shared memory arena are used by the draw callback to determine which method to use.

2. Intersection Testing

a. Collision Detection

A pre-traversal callback is used for the vehicle and munitions collision detection. This callback keeps track of the current entity node, so that if an intersection occurs, the identity of the entity that was hit can be returned to the application.

Collision detection is currently implemented using a segment intersection with the geometric primitives. The segment is created from the point that an entity is located at in the beginning of a frame, to the point the entity would be in at the end of the frame if it does not hit anything.

b. Vehicle Control

Several intersection tests are done for determining where to place the driven vehicle within the world. If the driven vehicle is a truck, then each time the vehicle is moved, the simulation must ensure that it remains on the ground, not above or below it. If the driven vehicle is an aircraft, you need to be able to land on the ground if you have the appropriate orientation to do so, but crash if you don't.

The X and Y coordinates of the vehicle's position are used to create a segment going straight down from the vehicle (in some cases from above the vehicle) to determine the Z coordinate of the terrain. If the entity's Z coordinate is at or below the ground Z, a ground orient function is called. This function takes the normal to the terrain polygon that was intersected, and with the entity's heading, calculates what the proper pitch and roll should be for the vehicle at that location.

VI. AUTONOMOUS AGENTS

A. INTRODUCTION TO CLIPS

CLIPS was chosen as the inference engine for this system for a variety of reasons. The foremost reason being that CLIPS can be easily and effectively interfaced with C. The second is that CLIPS is designed for forward chaining, which means that it makes decisions based on a set of current facts. In a combat simulation, autonomous entities need to be able to respond to the current situation in a specific way. For NPSNET-IV, we wanted entities that could conduct some pre-assigned mission, and then, if the entity receives data that certain conditions have been met, it responds in some predetermined manner.

There are at least three different ways to build an expert system with CLIPS and C. CLIPS can act as the outer shell and make calls to C functions, a C program can act as the outer shell and call CLIPS subroutines for decision making, or a hybrid of the two previous methods can also be used [NASA91]. For NPSNET-IV, we chose to use CLIPS as the outer shell. This choice was based simply on the previous success of other students using this method [SCHM93].

CLIPS uses a set of facts and a set of rules to determine what actions to take. The rules are broken into two parts, the Left Hand Side (LHS) and the Right Hand Side (RHS). Each rule is similar to an if (expression) then (statement) construct, with the expression corresponding to the LHS and the statement corresponding to the RHS of the rule. CLIPS utilizes pattern matching between the LHS of each rule and the current facts list to decide which rules are added to the agenda. After all the rules have been checked, the last rule to be placed on the agenda is fired, which means all of the statements on its RHS are executed. When the statements on the RHS are executed, the facts list may be altered, which may effect the rules added to the agenda the next time they are checked.

After a rule is fired, it is removed from the agenda. When the agenda is cleared, all of the rules are again checked for matches with the new facts list. For a rule to fire again, one of the facts in the fact list that matched the LHS of the rule must be changed, or the rule must be explicitly reset.

If a rule were to retract and then reassert some fact without making any other changes, the code would go into an endless loop because the same rule would end up being the last rule on the agenda each time. On the other hand, if facts are not removed from the facts list, the program would get slower as the facts list grows, or just stop execution because there is no rule that can be fired. See [Giar91] and [NASA91] for a more detailed description of the CLIPS language

B. NPSNET AUTONOMOUS VEHICLE CONTROL CLIPS SHELL

Figure 13 illustrates the overall block diagram for the NPSNET Autonomous Vehicle Control (NPSNET-AVC) program flow. From outward appearances, you can not tell that NPSNET-AVC is running with CLIPS as the outer shell. We developed it with an executable, 'autonomous', that takes configuration files as command line arguments. These configuration files are the ones that are used with the NPSNET-IV simulator code.

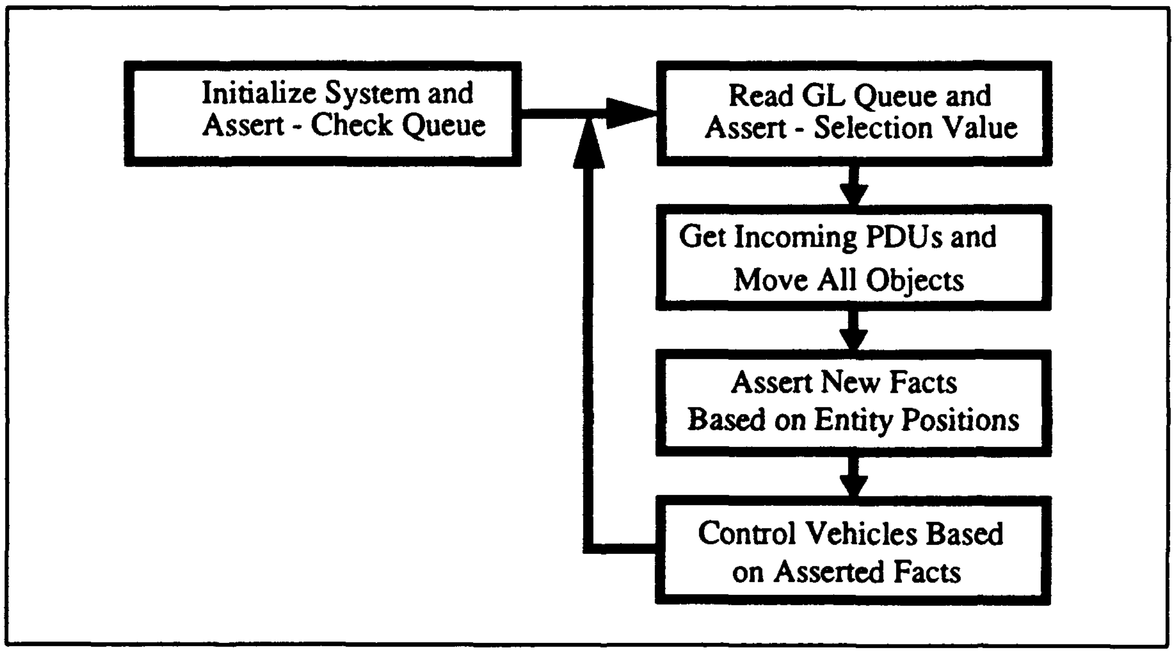
Initialize System and Assert - Check Queue
Read GL Queue and Assert - Selection Value
Get Incoming PDUs and Move All Objects
Assert New Facts Based on Entity Positions
Control Vehicles Based on Asserted Facts

Figure 13 CLIPS Block Diagram

When the code is executed, the CLIPS environment is initialized first, then all of the CLIPS rules are loaded, a CLIPS reset command is executed, and finally a CLIPS run command is executed.

The first CLIPS rule that fires calls the C++ initialization procedure. This procedure creates the array structures used to hold the entity records, the array structure for entity types, and a weapons array. The configuration, terrain, dynamic models, and static models files that are used by the NPSNET-IV simulator are used by NPSNET-AVC.

1. 'C' to 'C++' interface

The process of interfacing functions and passing data between CLIPS and C is a straightforward procedure, but since NPSNET-IV is written in C++, we wanted to use C++ instead of C. Unfortunately, the CLIPS to C++ interfacing was not as straightforward. Because CLIPS is setup to handle function calls made to C and not C++, we decided the easiest way to integrate the two together would be to build an additional layer into the function interface. The CLIPS code makes a call to a C function, the *Clips Reference Manuals* [NASA91] provide the details and examples on the mechanics of building this interface. We encapsulated all of the C functions that the simulator uses into a single file called 'c_to_C_interface.C'. Each of the C functions simply call their C++ counter part with the same parameters that were passed out by CLIPS. The C++ function return values are then returned to the C functions which just pass the values on into CLIPS.

A special C++ compile switch, -F, is used to compile the 'c_to_C_interface.C' file and the results are redirected into a file named 'c_to_C_interface.c'. This file is in turn compiled using the standard C compiler to create the object code. This two step process of creating the object file for the interface allows the compiled C function calls to be able to call the name-mangled C++ functions.

2. System Control Rules

After the initialization is done, a check queue fact is asserted. This fires a CLIPS rule that calls a check_queue function. If nothing is found on the input queue, a continue loop rule fires. This rule then calls the following C functions:

- get_messages - gets the PDUs from the network.
- move_objects - moves all of the active autonomous vehicles.
- dr_objects - dead reckons all active entities in the vehicle array.
- ship - updates the ships location, see [Hear93] for details on this function.
- closest_plane - gets the local id for the closest plane to the Loch Ness monster.
- entity_position ?*NESSY_ID* - gets the Loch Ness monster's position.
- entity_position ?*DRONE_ID* - gets the drone aircraft's position.

The last three C function calls assert positional facts into the CLIPS facts list. These facts are then used by the clips rules to determine what the entity should do next, based on the available facts. After the decisions have been made by CLIPS, the cycle starts over again. When a selection is found on the queue, one of the entity control rules maybe called, the system maybe reset, or the system maybe terminated.

3. Entity Control Rules

The entity control rules are grouped into four major areas. The first group is the start entity rules, which give the entities an initial posture and velocity or acceleration. The second is the stop entity rules which are used when the entity is deselected from the menu, or the system reset is invoked. The third group is for controlling the Loch Ness monster when a low flying aircraft comes within the monster's specified area of operation. The fourth group of rules is for maintaining, or changing the direction that an entity is moving, based on the current facts list.

VII. NETWORK IMPLEMENTATION

A. BACKGROUND

Because of the costs of live training exercises, simulation systems are going to play an ever increasing role in military training. For these simulators to be effective, they must provide realistic scenarios, react to the operator's inputs in real-time, and they must be accessible. In order to achieve the complexity and realism required to keep even one participant submerged in a believable training simulation, requires a great deal of computational power, and to have an interactive simulation requires even more. Even with the most powerful commercially available computing systems available today, a small number of entities interacting with each other in a realistically rendered virtual world environment can bring a single machine based simulation to a crawl.

One way to overcome this computational limitation, is to break the large simulation system into smaller stand-alone simulation systems that interact with each other over a high speed network. This allows each smaller simulation system to handle the intersection testing and user inputs for just a few locally controlled entities. The remotely controlled entities can be dead-reckoned between the receipt of update messages from the network, relying on the other host workstations to pass on changes in their status.

NPSNET-IV is designed to control one dynamic entity at a time, and the ordnance that it delivers, on an SGI Indigo or Reality Engine workstation. It computes the posture and status of all locally controlled entities and broadcasts this data over a network using the DIS protocol.

B. DIS PROTOCOL

The DIS protocol, which is designed to replace the SIMNET protocol, is still under development. The only portion that has been accepted as a network protocol standard for the IEEE are the data packets used for simulator interaction across high speed networks like an ethernet. Each of these data packets is called a Protocol Data Unit (PDU). There are currently twenty-seven PDUs defined by the proposed standard, but only four of these are for entity interaction. The remainder of the PDUs are for transmitting information on supporting actions, electronic emanations, and for simulation control [IST93].

NPSNET-IV currently uses three of the four entity control PDUs, these are the Entity State PDU, the Fire PDU, and the Detonation PDU. The Entity State PDU is used to communicate information about a unit's current state, including position, orientation, velocity, and appearance. The Fire PDU contains data on any weapons or ordnance that are fired or dropped. The Detonation PDU is sent when a munition detonates or an entity crashes. The actual structure of a PDU is very regimented and is explained in full detail in [IST93]. The fourth entity interaction PDU has not been incorporated into NPSNET-IV. The Collision PDU is sent out when an entity collides with another entity or static object in the virtual world.

C. PLAYERS AND GHOSTS

The networking technique used in NPSNET-IV follows the *players and ghosts* paradigm presented in [Blau92]. In this paradigm, each object is controlled on its own host workstation by a software object called a Player. On every other workstation in the network, a dead-reckoning version of the Player must be present, and is controlled by an object called a Ghost.

The Ghost objects on each workstation update their own position each time through the simulation loop, using dead-reckoning. The Player tracks both its actual position and the position that it calculated by using dead-reckoning. A posture update is sent out on the network when the two postures differ by a certain amount, or when a fixed amount or time has passed since the last update. When the updated posture for a Ghost object is received, the Ghost's posture is set to the update values, and the Ghost begins to dead reckon from this new posture. [IST93]

The primary purpose of the Ghost is to prevent network overload through the use of dead-reckoning. If every workstation tried to send out an updated PDU after every frame, the net would be saturated with packets and the system would quickly come to a crawl. This paradigm allows accurate data to be displayed on each workstation without saturating the network through the use of dead-reckoning.

D. IMPLEMENTING DIS IN NPSNET-IV

In NPSNET-IV, each entity, local and remote, is created as an object. The locally controlled entities send out a PDU at least every five seconds, or whenever a significant change occurs. These changes include ordnance drops, detonations, changes in appearance, and out-of-limit differences between the entity's true posture and the remote entity approximated posture.

Entities that are being controlled remotely update their posture using the appropriate remote entity approximations between PDUs. When an update PDU is received, the remote entity's posture and remote entity approximation data are corrected to reflect the data received. The object then continues using remote entity approximation from the updated location, with the updated velocity and/or acceleration vectors.

Most of the DIS protocol implementation was very straightforward. However, because NPSNET-IV was developed using Performer, it uses an entity coordinate system that is different from the one used by DIS, and both of these are different from the coordinate systems used by GL and mathematical functions. Figure 14 shows the entity coordinate systems used by DIS and Performer. Both coordinate systems are right handed, which makes the conversion from one to the other a little easier, and the entity orientations within the two coordinate systems make it even easier once you see the relationship.

An entity in the DIS coordinate system is oriented with its front pointed along the positive X axis, its right side along the positive Y axis, and the positive Z axis is straight down. In Performer, an entity is oriented with its front pointed along the positive Y axis, its right side along the positive X axis, and positive Z is straight up. Since both systems also use the heading-pitch-roll (hpr) convention for defining the entity orientation, once the pitch and roll values are calculated, only a conversion to radians for broadcast to the network and back to degrees for Performer are required

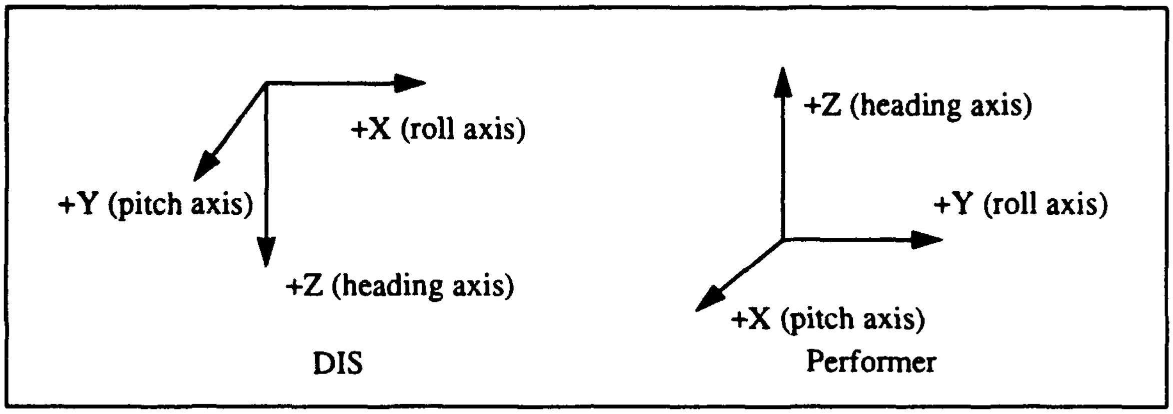

Figure 14 DIS and Performer Coordinate Systems

The heading values are not the same, but to go from one system to the other simply requires the heading angle to be negated, and converted to or from radians. In NPSNET-IV, the heading angle is always between zero and three-hundred sixty degrees, so all of the heading angles sent out on the network are negative. Figure 15 shows the relationship between Performer and DIS heading angles and the angles used for trigonometric functions.

The reason the conversions in Figure 15 work for trigonometric functions, is that the world coordinate systems for both Performer and DIS are the same. When an entity's location is given in one system, it is the same for the other system.

Part of the DIS network harness that was developed concurrently with this work is a set of network utility routines (netutils) that are used to decode the incoming PDUs. The netutils build a hash table based on the DIS EntityID that is used to determine if there is an active entity in the local simulator for each PDU received. If the entity is not found, an empty location in the vehicle array is located, an object is created with the empty location index as its local id. The NPSNET entity type and icon representation for new entities are then found by a tree traversal using the DIS EntityType. When the netutils has determined what object the incoming PDU is for, it then calls the appropriate member function for that object to process the PDU.

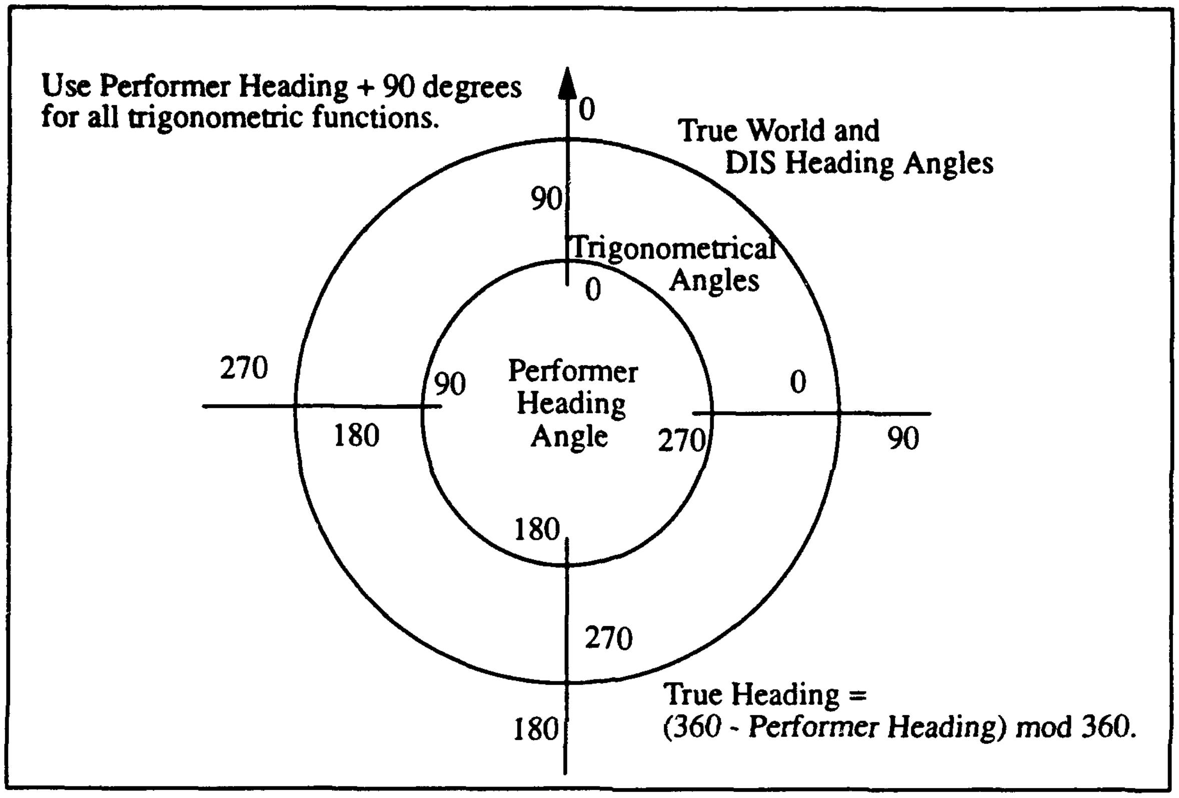
Use Performer Heading + 90 degrees for all trigonometric functions.
0
True World and DIS Heading Angles
90
Trigonometrical Angles
0
270
90
Performer Heading Angle
0
180
270
90
180
270
True Heading = (360 - Performer Heading) mod 360.
180

Figure 15 Heading Conversions

1. Entity State PDUs

The Entity State PDU is the largest PDU implemented in NPSNET-IV and is used to provide information about the entities that are participating in a simulation. It contains a lot of information that remains constant for a given object, such as protocol version, exercise ID, country, category, etc. The only fields that routinely change are position, orientation, velocity, and appearance of the entity.

a. Sending

The primary method for determining when to send an Entity State PDU is based on the Player and Ghost paradigm described in the section entitled "PLAYERS AND

GHOSTS". However, DIS uses the term "remote entity approximation" rather than "dead-reckoning" because dead-reckoning implies constant velocity, which is too limiting for modeling many types of entities.

The second method used is based on whether a predetermined period of time has elapsed since the last PDU was sent out. The DIS protocol established a default time limit of twelve seconds for removal of a remote entity from a simulation if no PDU is received for it. The NPSNET-IV implementation uses a predetermined time of five seconds maximum between PDUs. This allows for one PDU to be lost, allows for the network latency, and still provides more than a one second time buffer before a remote entity is removed.

Entity state PDUs are sent by the member function "sendentitystate()" of the VEHICLE and MUNITION base classes. These member functions fill in the required data for the PDU, and then pass it off to the net_write function of the DIS network library.

b. Receiving

After netutils determines which local object an entity state PDU is for, the member function "entitystateupdate()" for the VEHICLE or GROUND_VEH classes is called. An entitystateupdate() member function was defined for the derived class GROUND_VEH because the ground vehicles must follow the ground and can also have articulated parts like the turret and main gun on a tank. All remotely controlled entities are currently mapped to the VEHICLE class or a class derived from the VEHICLE class. The entity's posture data, velocity, dead-reckoning parameters, and appearance are updated to match the values in the PDU.

2. F[illegible] PDUs

Fire PDUs are used to notify other simulators that a munition has been fired. This allows other simulators to create an appropriate effect for the event. A good example is a muzzle flash when a main tank round is fired.

a. Sending

When an entity drops or fires some type of weapon, it calls a friend function from the MUNITION class to send out a fire PDU. This PDU identifies the entity that shot the munition, the location and orientation it was shot from, and has provisions to provide the entity id of the intended target.

b. Receiving

Currently the simulation code does nothing with a fire PDU when it is received. The NPSSSS does use the fire PDUs to determine when and where to make sounds appropriate to the weapons drop being made. With the advanced targeting techniques that are currently being developed for the system, the fire PDU may have a bigger part to play in the future.

3. Detonation PDUs

Detonation PDUs are used to notify other simulators when an explosion has occurred. The explosion can be the result of a munition going off, or from the crash of an entity.

a. Send

Detonation PDUs are sent by the member functions "senddetonation()" for the VEHICLE and MUNITION classes. All five of the senddetonation member functions call a member function "fillinDetonationPDU" to fill in the parts of the detonation PDU that do not change for a given entity.

A VEHICLE sends a Detonation PDU when it crashes into the ground or into static objects such as a building. This function is passed only the location of the detonation, which results in a fireball and smoke.

A MUNITION can call four different member functions, based on why the explosion occurred. The first case is a ground shot, and results in a crater, fireball, and smoke. The second is an explosion in the air due to the weapon reaching its max range, it results in only a fireball. The third is a when a building is hit, this results in a fireball and a model switch to a destroyed appearance, if appropriate. The last is when another entity is hit, this also results in a fireball, smoke, and a model switch.

b. Receive

When a detonation PDU is received, netutils calls the function "gotdetonate()". This function determines if the detonation occurred close enough to the locally controlled entity to damage or kill it, and creates the proper effects for where the detonation occurred. This function also sets the entity appearance to the appropriate status so that the entity knows what its behavior should be.

VIII. RESULTS, CONCLUSIONS, AND FUTURE WORK

A. RESULTS

1. Performance

The primary test for NPSNET-IV was a six day demonstration in the Tomorrow's Realities Gallery exhibition of the 1993 ACM SIGGRAPH, held in Anaheim, California, August 1 to 6. The system setup for the demonstration included nine workstations connected by a local network using thin Ethernet, and four remote workstations connected through a T1 line from NRad in San Diego. Two of the four remote workstations were at the ARPA Simulation Center in Arlington, Virginia. One workstation was located a NPS in Monterey, California and the last was at AFIT in Dayton, Ohio.

The NPSNET-IV simulator ran on five of these workstations throughout the demonstration. The frame rate on the 440 and Crimson Reality Engines ran between ten and twenty frames per second with a full screen display. The 440 was running with a single Raster Manager, and was limited to 20 frames per second by the system's pixel fill rate. The Crimson, a one processor machine, also had one Raster Manager. It was pixel fill bound, and the simulator was forced to run on a single thread. The ONYX was equipped with a second raster manager for the demonstration, and had no problem maintaining the requested thirty frames per second with a full screen display.

The NPSSSS and NPSNET-AVC were both run on Indigo workstations throughout the demonstration. NPS and AFIT had noise entity generation code sending out PDUs for twelve trucks and twelve planes from their home sites. AFIT also had their virtual cockpit and battle bridge, and sound server running on the other local workstations. This provided a total of around 35 entities constantly on the network. With munition drops and missile firing, the peak entity count was around fifty-three.

The terrain database used for the demonstration was 115 by 274 kilometers, and when loaded into performer, totals over 36,000 polygons. A 10 kilometer far clipping plane and sixty degree horizontal field of view was used.

2. Development and Extensibility

The actual development time for the current version of NPSNET-IV was six weeks with three people developing actual simulator code. A lot of the previous version was used in the development, but switching to the object oriented paradigm and setting up the underlying structure to support multiprocessing, required major modifications even to the portions that were used.

Upon return from the SIGGRAPH demonstration, we implemented a SAM site to shoot down Scud ballistic missiles into the NPSNET-AVC. The SAM site was developed using the same entity behaviors that the Loch Ness monster uses except extended to a third dimension. This implementation took five days, from concept to working demo. The addition of a ballistic missile class and development of a stand-alone system to control it took three days. Another change that was made to the code was the addition of a SUBMERSIBLE_VEH class. This class was needed to meet the peculiarities of underwater vehicles. The addition and testing of this new class took about thirty minutes.

B. CONCLUSIONS

The primary purpose of this work was to develop a new NPSNET simulator that would take advantage of the software and hardware technology that is currently available. NPSNET-IV takes maximum advantage of the advanced hardware features of the SGI graphics platforms available on the market today, allows the use of many user interface devices that the previous versions did not support, and is one of the first interactive distributed simulation systems to use the DI⌣ ρrotocol.

After performing the development, testing and evaluation of the various features of this project, we have reached the following conclusions:

- Use of the C++ object oriented paradigm created a more extensible NPSNET simulation system.
- Performer allows for rapid development of complex real-time virtual world applications.
- Spatialized sound greatly increased the feeling of 3D immersion.
- The partial implementation of the DIS protocol proved very successful in networking heterogeneous simulators both in a local environment and over a long haul network.

C. FUTURE WORK

This work provides a more extensible system from which to develop the ultimate goal of the NPSNET research group: a fully interactive and realistic virtual world simulation system. Some of the areas for future work are:

- Incorporating physically based motion into vehicle movement. Work being done by the vehicle control research group has already laid the groundwork for this with the development of the rigid-body class.
- Adding the reactive behaviors of noise entities from NPSNET-1 to the entities controlled by the NPSNET-AVC.
- Extending the use of LOD models, and developing converters to support the use of other database formats.
- Continue the implementation of the DIS protocol suite.
- Adding Head Mounted Display HMD
- Adding Graphic User Interface GUI

This work provides the basis for the future development of NPSNET. The possibilities for expansion are limited only by the imaginations of future researchers and hardware capabilities.

APPENDIX A. NPSNET IV USER'S GUIDE

NPSNET-IV

The Performer Based Version

Roy D. Young, Paul T. Barham, and David R. Pratt
Department of Computer Science
Code CS/Pr
Naval Postgraduate School
Monterey, California 93943
pratt@cs.nps.navy.mil
Fax: (408) 646-2814

Overview

NPSNET-IV is a DIS version 2.0.3 network compatible Silicon Graphics workstation based distributed, interactive, virtual world simulator. This document is designed to give the user the ability to start and run the NPSNET-IV system. For those of you who want to get started right away, you can invoke the program by "npsnetIV".

Please let us know what you think of the system. We will keep track of all known users and provide E-mail announcement of updates as they occur.

Capabilities

The terrain database, dynamic models, and static models used in NPSNET-IV were created using MultiGen Flight version 12. Collision detection has been implemented between the dynamic entities and all static features in the virtual world environment. Collisions between two dynamic entities is not implemented at this time.

NPSNET-IV is limited to 200 active entities at this time which is an arbitrary value set in the file "constants.h". After an entity is deactivated, its slot in the global vehicle array is available for reuse. If there is not a slot for the incoming vehicle the system reports the error and continues to process other PDUs.

The following DIS PDUs are implemented: Entity state, Fire, Detonation. All other PDUs are not implemented, and are discarded as they are received.

NPSNET-IV has several ASCII files that provide configuration information. This allows the use of different databases without the need to re-compile the program.

Directory Structure

NPSNET-IV directory structure is self contained, with the exception of the standard headers files located in /usr/include, and the Performer header files located in /usr/include/Performer. The Makefile contains only relative path addressing. In order for the code to compile, the workstation must have Performer loaded on the system.

The simulator code is located in ~trg/beta directory. This directory is set up with static links to all of the directories required to make and run the code from this directory.

The datafiles subdirectory contains most of the ASCII parameter files used to configure the system. These files can be changed without re-compiling the system. The NPSNET-IV vehicle types are defined in "dynamic.types" in the datafiles directory.

The file "static.loc" is used to identify what static objects should be loaded, and provides a posture for them. The name of the model is given, followed by the relative name of the directory the model file is in. This file is also contained in datafiles directory.

The DIS networking routines header files are in the network/h directory. The network code is in the network/code directory, and the network library is in network/bin.

The flight control interface code for both the Thrust Masters and the Kenny Aero is located in the FCS directory.

The font used for the HUD is located in the fonts directory. This directory also contains a font editor that can be used to customize the HUD symbols.

The autonomous vehicle controller (NPSNET-AVC) is located in the ~trg/clips directory. The executable for it is "autonomous" and can use the same configuration files as NPSNET-IV.

The spatialized sound system (NPSSSS) is located in the ~trg/sound directory. The executable for it is "NPSSSS" and has no command line arguments.

Command Line Arguments

In order to provide the maximum flexibility we have included the ability to use configuration files as command line arguments. There are no dashes (-) before any of the configuration file names. For example, to configure NPSNET-IV to start up with the driven vehicle as a truck, you could enter the following command: 'npsnetIV config.truck'. This would require a configuration file named 'config.truck' that contains at a minimum, the configure option VEHICLE_DRIVEN followed by the model name for a truck.

The following configuration options are available for designing a custom configuration file with the options you want. The numbers in parentheses indicates the minimum number of letters that must be used in the configuration file to identify the option. The configuration file options are listed below.

```
# COMMENT
BOUNDING_BOX(2) <xmin xmax ymin ymax>
CHANNEL(2) <channel_number>
CLIP_PLANES(2) <near far>
CONTROL(2) <KAFLIGHT(2)/FCS(1)/SPACEBALL(2)/KEYBOARD(2)>
EARTH_SKY_MODE(2) <FAST(1)/TAG(1)/SKY(3)/SKY_GROUND(10)/SKY_CLEAR(10)>
FCS_PORT(6) <portname>
FCS_PROMPT(6) <ON(2)/OFF(2)>
FIELD_OF_VIEW(2) <x_fov y_fov>
FOG(3) <ON(2)/OFF(2)>
FOV(3) <x_fov y_fov>
FRAME_RATE(2) <rate>
GROUND_HEIGHT(2) <elevation>
HEAD_MOUNTED_DISPLAY(2) <ON(2)/OFF(2)>
HMD(2) <ON(2)/OFF(2)>
HORIZON_ANGLE(2) <angle>
HUD_FONT_FILE(2) <filename>
INTRO_FONT_FILE(2) <filename>
KAFLIGHT_PORT(2) <portname>
```

```
LOCATION_FILE(2) <filename>
MASTER(2) <machine_name>
MODEL_FILE(2) <filename>
MULTISAMPLE(2) <ON(2)/OFF(2)>
NETWORK_FILE(2) <filename>
NOTIFY_LEVEL(2) <FATAL(1)/WARN(1)/NOTICE(1)/INFO(1)/DEBUG(2)>
PHASE(2) <FREE(2)/FLOAT(2)/LOCK(1)>
PIPE(2) <pipe_number>
PROCESS_MODE(2) <APPCULLDRAW(8)/APP_CULLDRAW(9)/APPCULL_DRAW(8)/
APP_CULL_DRAW(9)/DEFAULT(1)>
ROAD_FILE(2) <filename>
SOUND_FILE(2) <filename>
TERRAIN_FILE(2) <filename>
VEHICLE_DRIVEN(2) <filename>
VIEW(4) <AIRPORT(1)/CANYON(1)/VILLAGE(1)/PIER(1)/START(1)>
VIEWPORT(5) <xmin xmax ymin ymax>
VIEW_HPR(6) <h p r>
VIEW_XYZ(6) <x y z>
WINDOW(6) <SVGA(2)/VGA(1)/FULL(1)/NTSC(2)/TV(2)/PAL(1)/STEREO(2)/NORMAL(2)>
```

Setting up the Network Interface

The only thing required to set up the network is to have the file "hosts.data" in the datafiles subdirectory, and it must include the name of the workstation you wish to use, the DIS Id, and the Ethernet interface.

Terrain Database Specification

The TERRAIN_FILE listed in the configuration file, "trg.terrain" by default, is used to load the Flight models that make up the terrain. Each model to be loaded must be preceded with an 'F'. This is followed by the complete relative address of the file to be loaded, and then the intersection mask to be applied to the model.

Run-time Interaction

Once NPSNET-IV is running, you can control the dn.en vehicle by means to the flight controls, Spaceball, or keyboard. To tether the view point to another vehicle, simply select the vehicle from the 2D HUD overlay with the left mouse button. While you are tethered to a vehicle you can not use any of the vehicle movement controls. You can control the relative location of the eye point to the vehicle you are tethered to with the eight arrow keys and the page up and down keys. To un-tether the view point, hold the shift key and press the left mouse. The following is a list of active keys, and a description of what they do.

1	- Toggle absolute/relative HUD
2	- Toggle HUD color mode
3	- Toggle HUD icon mode
A	- Accelerate vehicle
C	- Toggle clouds on/off
Comma	- Decrease fog (<)
Ctrl, Pad Enter	- Master reset for program (panic)
D	- Decelerate vehicle
Delete	- Reset view position to vehicle position
Down Arrow	- (Paused) Move view position backward
Down Arrow	- (Unpaused) Decrease vehicle pitch
End	- Fire missle
Enter, shift	- Reset resting stick position
Equal	- Incrase eye separation for stereo (+)
Esc	- Quit program
F	- Toggle fog on/off
F01	- Toggle Performer statistics
F02	- Toggle vehicle altitude/speed primary information
F03	- Toggle vehicle/program secondary information
F04	- Cycle through heads-up display options
F05	- Toggle texturing on/off
F06	- Toggle wireframe on/off
F07	- Cycle through multisampline levels for antialiasing
F08	- Toggle stereo view (if stereo window and video format)
F09	- Reset vehicle position and orientation to the airport
F10	- Reset vehicle position and orientation to the canyon
F11	- Reset vehicle position and orientation to the village
F12	- Reset vehicle position and orientation to the lake/pier
Home	- Fire machine guns
I	- Change input devices
Insert	- Reset vehicle direction in view direction
Left Arrow	- (Paused) Move view position left
Left Arrow	- (Unpaused) Change roll to perform left bank
M	- Toggle measuring system, Metric <--> American
Minus	- Decrease eye separation for stereo (-)
Mouse Left Button	- Select new vehicle
Mouse Middle Button	- Transport to HUD X,Y location
N	- Narrow field-of-view (zoom in)
Pad 2	- Change view direction down
Pad 4	- Change view direction left
Pad 5	- Reset view direction in vehicle direction
Pad 6	- Change view direction right
Pad 8	- Change view direction up
Pad Minus	- Decrease radar range
Pad Plus	- Increase radar range
Page Down	- (Paused Only) Move view postion down
Page Up	- (Paused Only) Move view position up

Pause	- Pause/Unpause program
Period	- Increase fog (>)
PrintScreen	- Save screen into nps_screen.rgb
R	- Reverse stereo eye views
Right Arrow	- (Paused) Move view position right
Right Arrow	- (Unpaused) Change roll to perform right bank
S	- STOP
Scroll Lock	- Cycle text colors
Space Bar	- Drop bomb
Spaceball 1	- Rezero speed
Spaceball 2	- Reset view direction in vehicle direction
Spaceball 3	- Rezero roll (level winges)
Spaceball 4	- Rezero pitch
Spaceball 5	- Decrease spaceball sensitivity
Spaceball 6	- Increase spaceball sensitivity
Spaceball 7	- Fire Machine guns
Spaceball 8	- Rezero spaceball sensitivity and resting position
Spaceball Pick	- Fire missiles
Spaceball RX	- Change vehicle pitch
Spaceball RZ	- Change vehicle roll
Spaceball forward	- Increase speed
T	- Modify time of day
Tab	- Cycle HUD transparency amount
Up Arrow	- (Paused) Move view position forward
Up Arrow	- (Unpaused) Increase vehicle pitch
W	- Widen field-of-view (zoom out)

Limitations / Known Bugs

The biggest limitation is that you must be on an Indigo or Reality Engine Series workstation to run the code. NPSNET-IV uses run-to-completion frame updates. As a result, the more complex the rendered scene is, the lower the frame rate. The system is limited to 200 active entities. This limitation is created by a MAX_VEH constant in constants.h.

Acknowledgments

We wish to acknowledge the sponsors of our efforts, in particular LtCol. David Neyland, USAF, ARPA/ASTO. Sponsoring organizations: ARPA/ASTO, USA AI Center, USA TRAC, USA STRICOM, and the NPS Direct Funding Program.

LIST OF REFERENCES

[Blau92] Blau, Brian, Hughes, Charles E., Moshell, J. Michael and Lisle, Curtis, *"Networked Virtual Environments", Proceedings of the 1992 Symposium on Interactive 3D Graphics*, 30 March, 1992, pp. 157-164.

[Broo75] Brooks Jr, Frederick P., *The Mythical Man-Month*, Addison-Wesley Publishing Company, Philippines, 1975.

[Brys93] Bryson, Steve, van Dam, Andries, Pausch, Randy, Robinett, Warren, *Implementing Virtual Reality*, SIGGRAPH '93 course notes #43, SIGGRAPH '93, Anaheim, CA, 1993

[Corb93] Corbin, Daniel, *NPSNET: Environmental Effects for a Real-Time Virtual World Battlefield Simulator*, M.S. Thesis, Naval Postgraduate School, September 1993.

[Giar91] Giarratano, Joseph C., *CLIPS User's Guide, Vols 1 & 2*, Software Technology Branch, NASA - Lyndon B. Johnson Space Center, Houston, TX, January 1991.

[Hear93] Hearne, John, *NPSNET: Physically Based, Autonomous Naval Surface Agents*, M.S. Thesis, Naval Postgraduate School, September 1993.

[Henn90] Hennessy, John L., Patterson, David A., *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[IST93] Institute for Simulation and Training, *"Standard for Information Technology-Protocols for Distributed Interactive Simulation Applications (Proposed IEEE Standard Draft)"*, IST-CR-93-15, University of Central Florida, June 1993.

[NASA91] Software Technology Branch, *CLIPS Reference Manual, Vols I - III*, NASA - Lyndon B. Johnson Space Center, Houston, TX, January 1991.

[SCHM93] Schmidt, Dennis A., *NPSNET: A Graphical Based Expert System To Model P-3 Aircraft Interaction With Submarines and Ships*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, June 1993.

[SGI91] Silicon Graphics, Inc., *Graphics Library Programming Guide*, Document Number 007-1210-040, Mountain View, CA, 1991.

[SGI92a] Silicon Graphics, Inc., *IRIS Performer Programming Guide*, Document Number 007-1680-010, Mountain View, CA, 1992.

[SGI92b] Silicon Graphics, Inc., *IRIS Performer Man Pages*, Document Number 007-1681-010, Mountain View, CA, 1992.

[Soft92a] Software Systems, *MultiGen Modeler's Guide*, revision 12.0, San Jose, CA, 1992.

[Soft92b] Software Systems, *MultiGen Flight Format Modeler's Guide*, revision 12.0, San Jose, CA, 1992.

[Soft92c] Software Systems, *MultiGen Texture Option User's Guide*, revision 12.0, San Jose, CA, 1992.

[Zesw93] Zeswitz, Steven, *NPSNET: Integration of Distributed Interactive Simulation (DIS) Protocol for Communication Architecture and Information Interchange*, M.S. Thesis, Naval Postgraduate School, September 1993.

[Zyda92] Zyda, Michael J., Pratt, David R., Monahan, James D., and Wilson, Kalin P., "NPSNET: Constructing a 3D Virtual World.", *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, 30 March, 1992, pp. 147-156.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

2. Dudley Knox Library 2
Code 52
Naval Postgraduate School
Monterey, CA 93943-5002

3. Dr. Ted G. Lewis 1
Code CS/Lt
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000

4. Dr. Michael J. Zyda 1
Code CS/Zk
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000

5. Dr. David R. Pratt 4
Code CS/Pr
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000

6. Captain Roy D. Young 2
Studies and Analysis Branch
Marine Corps Combat Development Center
Quantico, VA 22342

7. LtCol David Neyland 1
Advanced Research Projects Agency / ASTO
3701 N. Fairfax Drive
Arlington, VA 22203

8. Director, Training and Education 1
MCCDC, Code C46
1019 Elliot Road
Quantico, VA 22134-5027

9. Mr. Paul Barham 1
Code CS
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000